

# OVERLAY TECHNIQUES FOR BLOCK STRUCTURED LANGUAGES

A Thesis Submitted  
in Partial Fulfilment of the Requirements  
for the Degree of  
MASTER OF TECHNOLOGY

By  
RAJEEV SEONI

*to the*

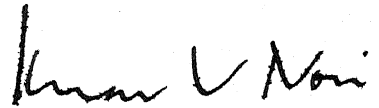
DEPARTMENT OF ELECTRICAL ENGINEERING  
INDIAN INSTITUTE OF TECHNOLOGY KANPUR  
AUGUST 1980

CERTIFICATE

This is to certify that the work entitled,  
'OVERLAY TECHNIQUES FOR BLOCK STRUCTURED LANGUAGES',  
has been carried out by Lt. Rajeev Seoni under our  
supervision and has not been submitted elsewhere for  
the award of a degree.



( V. RAJARAMAN )  
Professor  
Electrical Engineering Department  
and  
Computer Science Program  
Indian Institute of Technology  
Kanpur



( KESAV V. NORI )  
Assistant Professor  
Computer Science Program  
Indian Institute of  
Technology, Kanpur

EE-1980-M-SEC-CVE

I.I.T. KANPUR  
CENTRAL LIBRARY  
Acc. No. **A** 63792:

20 NOV 1980

To

my Father and Mother,  
and my sister Rashmi.



## ACKNOWLEDGEMENTS

I would like to take this opportunity to express my deep gratitude to Prof. Kesav V. Nori and Prof. V. Rajaraman (my thesis guides) for their help, guidance and encouragement.

I am thankful to Dr. S. Biswas and Mr. V.M. Malhotra, and to my colleague Mr. A.K. Dey, for their valuable suggestions.

My special thanks to Vandana for her corrective criticism of this thesis, and for all the help in proof-reading and preparation of slides.

My thanks to Mr. J.S. Rawat for typing and Mr. Ganga Ram for cyclostyling.

Rajeev Seoni

## ABSTRACT

This thesis is concerned with the following two problems:

- i) How can we decompose a large block structure program into smaller related components such that the resultant components can be used in Planned Overlay Schemes of memory management?
- ii) What are the requirements of a separate compilation facility for block structured languages?

We have approached both these problems through PASCAL, an example block-structured language. As a test case, we have used the PASREL compiler and have obtained its decomposition that suits our purpose.

## TABLE OF CONTENTS

Chapter		Page
1	INTRODUCTION	1
2	OVERLAYING AND GENERAL REQUIREMENTS FOR TREE-STRUCTURED OVERLAYING LOADERS	9
3	OVERLAY TECHNIQUES FOR BLOCK STRUCTURED PROGRAMS	15
4	GRAPH THEORY APPLIED TO OVERLAY TREES	22
5	APPLICATION TO A RECURSIVE DESCENT COMPILER	33
6	CHANGES IN PASREL TO SUPPORT OVERLAYS	34
7	CONCLUSIONS AND SUGGESTIONS FOR FURTHER WORK	36
	REFERENCES	38
Appendices		
A	PROGRAM LISTING	A-1
B	FINAL RESULT	B-1

## CHAPTER 1

### INTRODUCTION

In any computer system, efficient memory management is very important so as to obtain high efficiency of resource utilization and a satisfactory performance in the overall aims of the computer installation. The technique chosen for memory management should be closely dependant on the type of the work load of the computer and on the characteristics of the hardware available. Now, since the physical memory and even the virtual memory has some limits, it is possible that a program may be so large that it just cannot fit into the memory all at one time. Thus large programs cannot be run until and unless they are broken up into smaller parts in some way.

Our interest in this problem area arose because we were interested in transferring a large PASCAL compiler onto the local TDC-316 system.. A cross-compiler for PASCAL for PDP-11 running on the DEC system -10 was the starting point of this study. Unfortunately, the version of PASCAL on which this compiler could be cross-compiled was not available locally. We then became interested in segmenting the program into small segments so that it could fit in the limited memory resources of

TDC 316. Two problems that arose from this effort were:

- i) How do we generally break a large block structured (PASCAL) program into small parts so as to run it on small machines using simple memory management techniques?
- ii) How do we independantly compile procedures (arbitrarily nested) in a block structured language?

These two problems are tackled in this thesis.

### 1.1, Some Methods of Memory Management:

To execute programs too large for a certain computer system what is required, essentially, is to keep only those parts of the program in main memory which are required at that time. A reference to a part of the program not in main memory may cause the replacement of a part in memory by the referenced part.

At this point, it would be useful to clearly recognize that a program has its logical requirements of instruction code and data space. The logical requirement is met by mapping the instructions and data of the program into an address space, which may be

virtual or physical. Considering that the program itself is possibly constructed from several components with their own independent logical requirements, the basic problems in constructing the above mapping are relocation and linkage respectively.

Relocation causes a translation (shift) of some addresses used in the program so that the program (or even a component of a program) is consistent in its use of the address space with respect to a new origin.

Linkage problems are inherently concerned with the resolution of references across component boundaries (after the relocation problem has been tackled).

All memory management schemes are concerned with the above two issues. They differ only in the time, with respect to the execution of the program, at which the above problems are resolved.

Some of the methods of memory management available are, very briefly, as follows [2,3]:

(i) Planned Overlay: Overlaying is a technique where parts of a program are held on some external (or secondary) storage device and brought into main memory as required. In planned overlay, segments of the program are identified which need not be in main memory

together. The relationship of program segments also have to be planned in advance by the programmer. This method will be discussed in detail later.

In planned overlay, all relocation and linking is resolved before execution starts.

(ii) Dynamic Overlay: In this method no pre-planned overlay structure is required. As and when program segments are required, they are brought into main memory by explicit calls to the linking loader by the programmer. In certain cases, segments of the same program may be executed in parallel, in others, only serially.

(iii) Paging: In this scheme the main memory is divided into fixed length 'page frames', and each program into same length 'pages'. There are various sub-schemes for paging, which differ in when pages are brought into and when they are removed from main memory.

(iv) Segmenting: It is similar to paging, except that the programs are divided logically into variable length 'segments'. The segments may be further divided up into pages. Segmentation without paging is similar to Dynamic Serial Overlays.

## 1.2 Situations that are suitable for Planned Overlay:

In the case where the virtual memory is small or the physical memory is equal to the virtual memory, large programs will have to be broken up into an overlay structure to reduce their requirement of memory at any given time.

Programs that can be logically divided into major sections are well suited for planned overlay execution [ 2] . Also if the program structure follows well defined rules, planned overlay is suitable.

Planned overlay structures can be more efficient in terms of execution speeds compared to dynamic overlays because the linkage editor procedure permits direct references by one segment to values whose locations are identified by external symbols in another segment. There is no need to collect such values in a consolidated parameter list. Planned overlay optimizes the use of main memory, has lesser run-time overhead in comparison with Dynamic Overlay or Paging because there is no need for performing relocation, linkage or map table maintenance.

These advantages tend to diminish as the users' programs get more and more complex, particularly when



the logical selection of subprograms depends on the data being processed [2]. In this case, Dynamic Overlays seem better. A combination of both Planned and Dynamic Overlay structures may also be used. A module linked dynamically may itself operate in the planned overlay mode, and within a planned overlay program one may include dynamic overlaying.

### 1.3 Memory Requirements for Block Structured Languages:

Memory required by a program written in a block structured language can be classified as follows [4] :

i) Global data: This is permanently allocated for each program.

ii) Local data of procedures: This is usually allocated on the run-time stack on procedure call and deallocated on return from the procedure.

iii) Dynamically created data: This is usually allocated by a heap mechanism with some kind of garbage collection.

iv) Program code

Using Planned Overlay structures it is possible to overlay program code but not data. This is because data in the stack is dynamically created and referenced with respect to a base that is dynamically ascertained.

#### 1.4 PASCAL Implementation :

The implementation of PASCAL involves the running of the PASCAL compiler, which itself is a large program. If the compiler is a one-pass type, it has to perform the whole lot of functions involved in compilation at one go. Thus it naturally becomes very large, and implementation of PASCAL on mini and micro computers is not possible due to memory constraints. If the compilation could be broken into a number of phases, then parts of the compiler can overlay each other, thereby reducing the total memory requirement. The phases could be scanning the input a number of times, each time performing a small task, or also having one phase produce an output which could be the input to the next phase. By increasing the number of phases, the compiler can be divided into smaller and smaller sections which may overlay each other, enabling the implementation of PASCAL on mini and micro computers.

#### 1.5 Structure of the Thesis:

The first step in this thesis was the study of some tree - structured planned overlay systems, which is given in Chapter 2. In Chapter 3 are given a few algorithms which were developed and worked out for finding overlay tree structures, which did not give

satisfactory results, for block structured languages. Chapter 4 formalizes the basic problem in the construction of overlay trees for block-structured programs. Here, we apply the idea of strongly connected components of digraphs to the call graph of a program and obtain the minimally constrained Overlay Tree. A possible partition for the PASREL compiler, according to the algorithm which was implemented, is given in Chapter 5. Finally, in Chapter 6, the requirements for independant compilation facility for block structured languages to permit use of overlay techniques, are given. The last Chapter 7, contains the conclusions drawn from the work done for this thesis.

## CHAPTER 2

### OVERLAYING AND GENERAL REQUIREMENTS FOR TREE-STRUCTURED OVERLAYING LOADERS

Essentially, the technique of overlaying involves the division of large programs into smaller parts such that the parts are held on some secondary memory and brought into main memory only when they are required. Thus different parts of a program may occupy the same area of memory at different times. Generally, the routines of a program are grouped into a permanent unit and a number of overlay units (or nodes), and the available memory is divided into a permanent area and one or more overlay areas [1]. The permanent area holds the permanent unit as well as the non-overlaid data areas, which provides a communication area for overlay units occupying different overlay areas. Each overlay area holds, at any one time, one of a specified list of overlay units. This means that two units allocated to the same area cannot be in memory simultaneously.

#### 2.1 Overlaying:

To enable overlaying, the assembler must provide pseudo - operations by which the programmer can indicate how the program is divided up, that is, which routines are in one overlay unit, which overlay units

occupy the same memory area and which routines are in the permanent area. This information has to be passed on by the assembler to the linkage editor and loader.

The implementation of the overlaying technique involves a number of stages [1] . In the first stage, each routine is compiled separately and each overlay unit is linked like a complete program. Its storage requirements are evaluated and cross-references filled in. The difference is that calls to other routines have their addresses flagged as relative to the start of their own overlay area (unknown at this time) or to the start of the permanent area.

The second stage determines the size of the overlay areas. The size of an area is obviously that of the largest overlay unit that will occupy it. Once the sizes of all overlay areas have been determined, memory can be allocated and the origin for each overlay area determined.

The last stage is loading. One-by-one, each overlay unit is processed, being relocated according to the origin previously calculated, and the resulting binary is outputted to the secondary store. At this stage, it is not necessary to load a unit into the memory area in which it will be executed.

Finally, the permanent area is set up, and a table is made which gives the secondary storage address of each overlay unit and its associated overlay area. The program which reads and writes overlay units (called the Overlay Handler) is incorporated into the permanent area by the usual library mechanism.

## 2.2 Requirements of a Tree-Structured Overlaying Loader:

Let us now consider the overlay facility available with the LINK-10 Linking Loader of the DEC system-10 [6]. The overlay program has a tree structure. The nodes of the tree are called links, each of which contains one or more program modules. The links are connected by paths.

The top node of the overlay tree is called the root link, and it contains the permanent overlay unit, that is, the main program, the Overlay Handler, the non-overlaid data areas and such procedures in the program which are required to be present in main memory throughout the execution of the program. Below the root link are the first-level links, each of which is connected to the root link by one path. The level of the links increases as we go further down away from the root link. A link at level  $n$  is connected by a path to exactly one link

(the father link) at level  $n-1$ . It is obvious that a link can have more than one downward path (to successor links), but only one upward path (to ancestor links).

An overlay tree structure with six links is shown in Fig. 1. The code in a given link can make reference only to memory in links along a direct upward or downward path, i.e. in a link which is vertically connected to it. Thus, the link C can reference memory in itself, in the root link A, or in its successor links D, E and F. A reference to memory in B from C would be illegal.

In the overlay tree, all nodes at any one level overlay each other. In the tree shown, B and C overlay each other, and only one amongst D, E and F can be in main memory at one time. One more type of reference that is not allowed, but may arise due to recursion, is a call from C to E if it is possible for F to call C. This is because, once F calls C and C in turn calls E, E would try to overlay F which has not yet finished execution.

Due to the restrictions in memory references, only one complete (at most) vertical path is required in

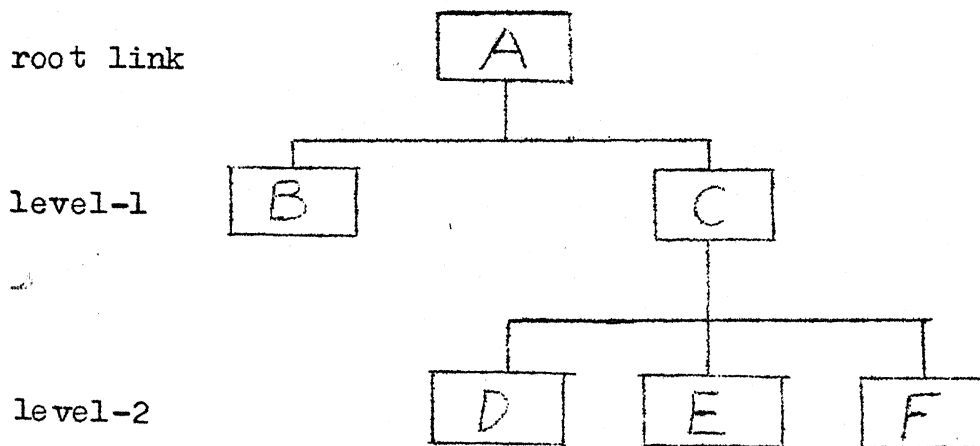


Fig. 1: An overlay tree structure

the virtual address space at any one time. The remaining links can be stored on a disk till they are required.

### 2.3 Program to Supervise Overlays:

There is a program which supervises the execution of an overlay program. Whenever a reference is made to a link which is not in memory, this program brings in the link, possibly overlaying one or more links already in memory. In the DEC system-10, this program is called the Overlay Handler [6]. The Overlay Handler is loaded into the root link, so that it is always in memory and can supervise overlaying operations from there.

Using the Overlay Handler, there are two ways of overlaying links during executions:

1. Implicit overlaying: A reference to a link not in memory implicitly calls the Overlay Handler to overlay one or more links with the required links.



## CHAPTER 3

### OVERLAY TECHNIQUES FOR BLOCK STRUCTURED PROGRAMS

The Block Structure Tree of a program is available to start with. Using the information of calls to and calls by each procedure, an overlay tree is to be constructed keeping in mind the requirements given in Chapter 2.

There are basically two ways in which the technique of overlaying can be used for reducing the memory requirements of a program:-

i) Programmed (or dynamic) overlaying of nodes:

This method uses the facility of the Overlay Handler by which the programmer can explicitly cause the overlaying of overlay units (links). No re-structuring of the program is necessary, but the programmer has to insert calls to the Overlay Handler at the appropriate places in the program. That means that the programmer has to keep track of the amount of memory used up by the program at different times during execution and if it crosses a certain limit, he must overlay certain units. At this point he must be careful that he does not overlay a calling procedure, i.e., one which has been executed only partially and control has passed out of it due to a

call by that procedure. The programmer will have to use the static link information to decide which overlay units will have to be brought in when a call is made to a procedure not in main memory.

This method is cumbersome and messy, and demands too much of work by the programmer. As far as restructuring and division of the program into overlay units is concerned, it is a trivial case.

(ii) Automatic overlaying: In this method, the Overlay Handler is called implicitly whenever a call is made to a procedure which is not in the main memory. The Overlay Handler then brings in the called unit, possibly overlaying a number of overlay units, according to some pre-planned overlay structure. This overlay structure is so planned that it takes care of all the requirements of the overlaying loader, or the Overlay Handler in DEC-10, like not overlaying a calling node etc.

So the main problem in using overlays boils down to re-structuring of the Block Structure Tree so that it conforms to the requirements of an overlaying loader. An algorithm which could give such an Overlay Tree would be very useful. Therefore a number of algorithms were developed and manually worked out for a large block structured program.

The best overlay tree is the one for which the total amount of memory required is the minimum, and also which uses memory for the least time during execution. Therefore, procedures which are rarely called should not be very high in the overlay tree because the higher a node, more is the time it spends in memory. Also, the overlay tree nodes should be as small as possible, in terms of memory required. A number of overlay trees may be possible for the same program. The programmer can select the one which has the minimum total memory requirement.

### 3.1 Heuristic Algorithms for Constructing Overlay Trees:

The basic methodology adopted in the algorithms developed is as follows:

The main program body was invariably put into the root node. All the procedures/functions declared in the main program (on level 1) were initially grouped together into one node, which is at level 1 of the Overlay Tree. In one of the algorithms, only those procedures which are declared in, as well as called by, the main program were put into this node. Then this node was split up into a number of brother nodes, all children of the same root node. All those procedures which call

each other have to be kept in one node. Those which do not call, and are not called by, any procedure in the node under consideration were separated out into separate nodes. A procedure which is called by a number of procedures in that node was shifted up into the parent node, or up into a new node at an intermediate level with all the calling procedures being kept in nodes which are children of this new node.

The above is repeated for all the procedures already placed in the overlay tree, i.e., all procedures declared in a certain procedure are initially grouped together into a child node, and then this node is split. But now one more thing has to be considered - a call backwards or upwards from the node under consideration, say from procedure A to procedure B. This sort of a call may cause problems, because the called procedure, B, may in turn call another procedure, say C, which may cause the overlaying of the first procedure, A. In this case, either the finally called (C) or the initially calling (A) procedure along with the procedures on the path have to be moved up into the node which has the backward called procedure (B). In one of the algorithms, the two paths were merged into one. One more possibility exists for a backward call - that the backward called

procedure does not fall on the vertical path to the root. In that case, it has to be moved up in the tree till it falls on the route, or else the two nodes (calling and called) have to be merged. After this step, all the calls to and from the shifted procedures have to be re-considered to eliminate all possibilities of a calling procedure being overlaid.

In the algorithms where only those procedures which are declared in, as well as called by, a procedure were initially grouped together, procedures which were not called were included in the first node from which they are called. This has an effect of pushing some procedures down in the overlay tree, which is good.

One of the algorithms developed took the block structure tree and started splitting/joining nodes from bottom up, using the same rules as the other algorithms discussed above.

It was seen that the algorithms were becoming very complicated because calls by procedures do not follow any set rules or pattern. A large number of possibilities had to be considered. Even then, it was seen that the overlay trees obtained did not help in reducing the total memory requirements of large programs because one of the

vertical paths invariably became very long compared to the others.

The possibility of duplicating procedures to separate two paths was also considered. But ultimately it was felt that no automatic algorithm could be developed which would give an appreciable reduction in the total amount of memory required, or the time for which memory is required by a program. This was attributed to the numerous other factors involved.

Some of the factors responsible for making the construction of Overlay Trees more complicated are:

i) The number of times a procedure is called should affect its position in the Overlay Tree.

ii) The size of code of a procedure should also be considered.

iii) Average time of execution of a procedure has an affect on the time for which memory is required by it.

Considering all these points, it was felt that an automatic algorithm should just give the basic, essential division of a program (i.e. which procedures have to be in one node in all cases), alongwith all the calls information. Then it can be left to the programmer to

consider all the factors discussed and constructed an Overlay Tree. This sort of an algorithm is available if we adopt the Graph Theory approach, which is discussed in the next chapter.

LIBRARY KANPUR  
CENTRAL LIBRARY  
Acc. No. A 63792

## CHAPTER 4

# GRAPH THEORY APPLIED TO OVERLAY TREES

Before the application of Graph Theory to the construction of Overlay Trees is discussed, it would be useful to define those terms of Graph Theory which will be used in this thesis.

#### 4.1 Definitions [7]:

(i) Graph: It is a finite set  $(V)$  together with a prescribed collection  $(E)$  of unordered pair of distinct elements.

e.g. :  $V = \{1, 2, 3, 4, 5, 6\}$

$$E = \{(1,2), (2,3), (3,4), (4,5), (5,6), (6,2), (6,4)\}$$

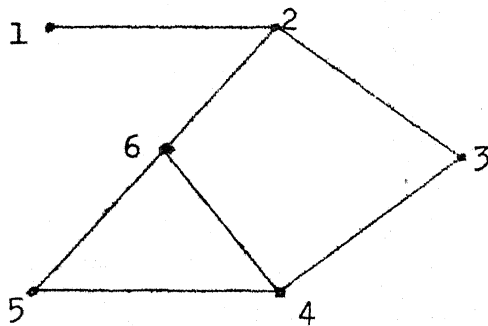


Fig. 2: An example of a Graph

(ii) Edge: Each unordered pair is called an edge, i.e., each element of  $E$  is an edge.

(iii) Vertex: Each element of the finite set (V) is called a Vertex. These are also known as Nodes of the graphs.



property if no larger subgraph contains it as a subgraph and has the property.

(xv) Strongly Connected (or Strong) Component: It is a maximal subgraph, of a digraph, in which every two points are mutually reachable.

#### 4.2 An Algorithm to Determine the Strongly Connected Components of a Digraph:

A digraph and its strongly connected components are shown in Figure 3.

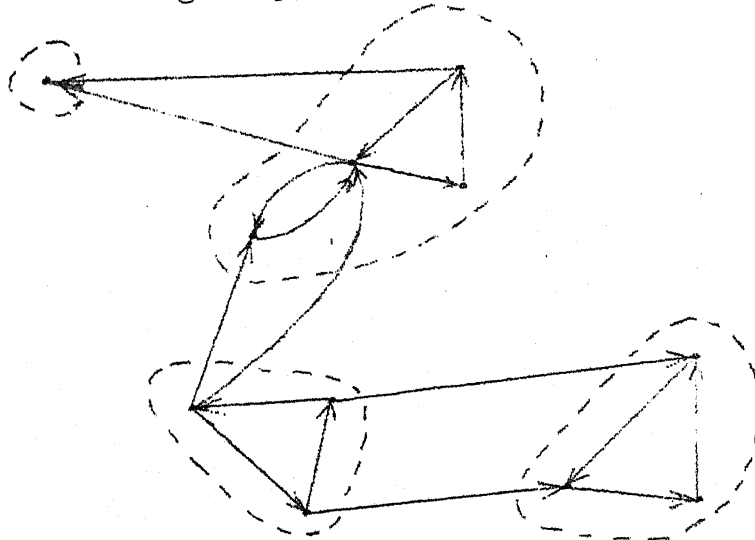


Figure 3: A digraph with its strongly connected components outlined by dashed lines.

Depth-first search can be applied to a digraph to determine the strongly connected components [5]. What is done in a depth-first search is this - one vertex ( $v$ ) is "visited", then one of the edges from  $v$  (say,  $(v, w)$ ) is

followed. If the vertex  $w$  has been previously visited, we return to  $v$  and choose another edge. If  $w$  has not been visited, we visit it and apply the process recursively to  $w$ . After all the edges leading out from  $v$  have been thus examined, we go back along the edge  $(u, v)$  that led us to  $v$  and continue exploring edges incident on  $u$ .

Let us consider what happens when we traverse the edges of a digraph  $G$  along their orientations during a depth-first search on  $G$ . We assign a serial number  $\text{nodeno}(x)$  to each vertex  $x$  the first time we visit it. If we encounter an edge  $(v, w)$  that has not been traversed, and  $w$  has not yet been visited, we mark this edge as a tree edge. If  $w$  has already been visited, then  $w$  may or may not be an ancestor of  $v$ . If  $w$  is an ancestor of  $v$ , then clearly  $\text{nodeno}(w) < \text{nodeno}(v)$  and  $(v, w)$  is a back edge. If  $w$  is not an ancestor of  $v$ , and  $\text{nodeno}(w) > \text{nodeno}(v)$  then  $w$  must be a descendant of  $v$  and the edge  $(v, w)$  is called a forward edge. If  $\text{nodeno}(w) < \text{nodeno}(v)$  and  $w$  is neither an ancestor nor a descendant of  $v$ , then the edge  $(v, w)$  is called a cross edge.

Figure 4(a) shows a digraph  $G$  which is represented by its adjacency structure shown, and figure 4(b) shows

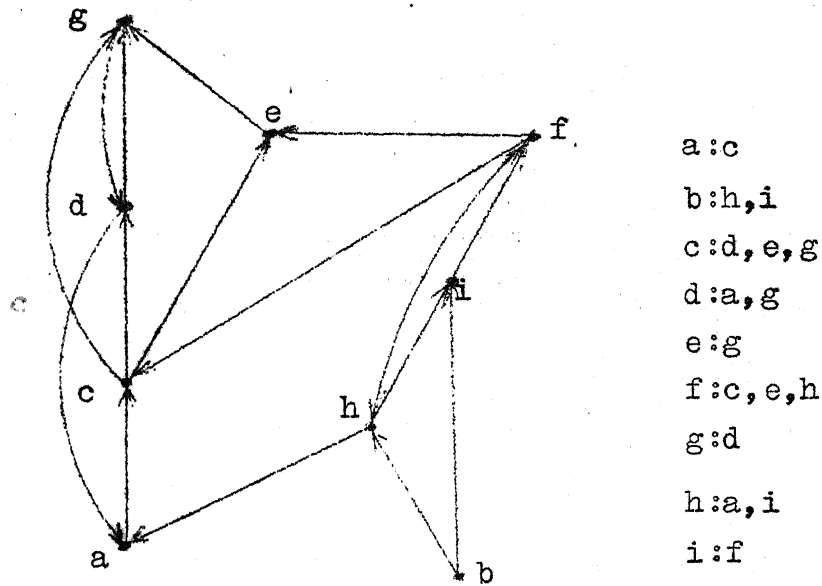


Fig. 4(a): A digraph G and its adjacency structure

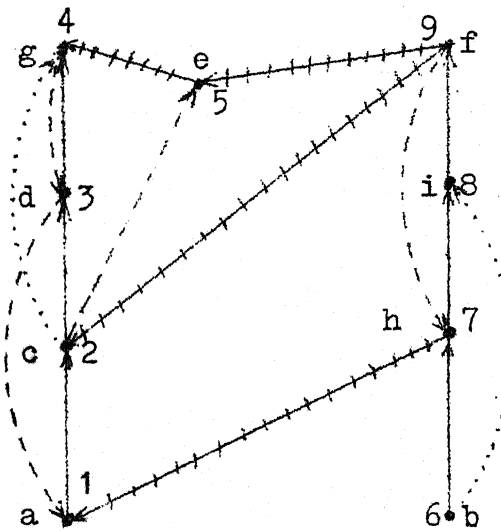


Fig. 4(b): Digraph G after a depth first search on it. The result consists of 2 trees (boldface), 3 back edges (dashed lines), 4 cross edges (crossed lines), and 2 forward edges (dotted lines).

the partitioning of the edges into four subsets as a result of the depth-first search. The numbers on vertices represent nodeno values.

Though the tree (or forest) generated by a depth - first search is not unique, it helps in determining the strongly connected components. Obviously, the forward edges can be ignored, since they do not affect strong connectivity. Also, both back and cross edges from  $v$  can only go to vertices  $x$  for which  $\text{nodeno}(v) > \text{nodeno}(x)$ . A little reasoning will show that if  $S$  is a strongly connected component of  $G$ , then the vertices of  $S$  define a tree which is a subgraph of the spanning forest.

To recognize the strongly connected components, we just have to identify the roots of ~~the~~ corresponding subtrees. To recognize these roots, we define enclblkno( $v$ ) to be the number of the smallest numbered vertex in the same strongly connected component as  $v$  that can be reached by following zero or more tree edges followed at most by one back edge or one cross edge. Thus,  $v$  is such a root if and only if  $\text{nodeno}(v) = \text{enclblkno}(v)$ .

An algorithm, taken from [5], using the above facts and determining the strongly connected components of a digraph, given by its Adjacency structure  $\text{Adj}(x)$ , is given below. We maintain a stack  $S$ .

```
procedure FINDCOMPS;
```

```
  begin
```

```
    i := 0;
```

```
    initialize S as an empty stack;
```

```
    for all vertices x in V do
```

```
      nodeno(x) := 0;
```

```
    for all vertices x in V do
```

```
      if nodeno(x) = 0 then STRONG(x)
```

```
  end;
```

```
procedure STRONG(v)
```

```
  begin
```

```
    i := i+1;
```

```
    nodeno(v) := i;
```

```
    enclblkno(v) := i;
```

```
    push v onto stack S;
```

```
    for all w in Adj(v) do
```

```
      if nodeno(w) = 0 then
```

```
        begin (*(v,w) is a tree edge*)
```

```
          STRONG(w);
```

```
          enclblkno(v) := minimum of (enclblkno(v),
```

```
            enclblkno(w))
```

```
        end
```

```
      else if nodeno(w) < nodeno(v) then
```

```
        (*(v,w) is a back edge or a cross edge*)
```

```

if w is on S then
    (*w is in the same strongly connected
      component as v, since w on S means
      there is a path from w to v *)
    enclblkno(v) := minimum of
      (enclblkno(v), nodend(w));
if enclblkno(v) = nodend(v) then
    (* v is the root of a strongly
      connected component *)
    while x, the top vertex on S,
      satisfies nodeno(x)  $\geq$  nodeno(v) do
      add x to the current strongly
      connected component and delete x
      from S.

end;

```

#### 4.3 Call Graph and Overlay Tree Generation:

If each procedure/function in a program is thought of as a vertex and each call from one procedure to another as a directed edge, we have a digraph which is known as a call graph. Once we obtain the call graph, the first step towards generating the Overlay Tree would be to determine the strongly connected components of this graph. Now it is obvious that all the procedures in one strongly

connected component have to be kept in one node (or link) of the Overlay Tree. The last section shows that automatic determination of strongly connected components is quite simple. As pointed out in the last chapter, having generated this essential division of a program, we leave it to the programmer to construct the Overlay Tree considering the numerous other factors involved.

#### 4.3.1 Call Graph Generation:

Generation of the call graph posed very interesting data structuring and procedure-name-table maintenance problems. We started with a simple lexical analyser which processed a subset of PASCAL (it was later modified to process the complete PASCAL).

We declared the following data structures:-

```

type
  PROCPTR = ↑ PROCNODE;
  LISTOFPROC = ↑ NEXTPROC;
  NEXTPROC  = record
      PROC : PROCPTR;
      NEXT : LISTOFPROC
  end;
```

```

PROCNODE = record
    NAME : ALPHA;
    LLINK, RLINK : PROCPTR;
    case ISPROC : boolean of
        true: (DECLPROC:PROCPTR;
                CALLS, CALLED BY,
                STRONGCOMP, LISTOFPROC;
                ONCSTACK: boolean;
                NODENO, ENCLBLKNO: integer)
    end;
var
    DISPLAY: array [0..LEVMAX] of record
        PROCS : PROCPTR;
        CURRENT : PROCPTR
    end;
    TOP : 0..LEVMAX;

```

The procedure-name-table is maintained in DISPLAY [TOP]. PROCS as an unbalanced binary tree at each declaration level. While processing a procedure/function declaration, a call is made to procedure ENTERPROC which creates a new PROCNODE, initializes all its fields and enters it in the appropriate place in the name-table. Then ENTERPROC also increments TOP, assigns the pointer to the procedure we have just entered to DISPLAY [TOP]. CURRENT, and initializes DISPLAY [TOP]. PROCS in anticipation of procedure declarations within this procedure.



On finally coming out of a procedure block, the `DISPLAY [TOP] . CURRENT↑.DECLPROC` is assigned the value of the pointer to the name-table (tree) of procedures declared in it, i.e., `DISPLAY [TOP] . PROCS`, and then `TOP` is decremented.

While processing the body of a procedure or function, whenever a call to another procedure or function is encountered, a call is made to procedure `ENTERCALL` which enters the call in both the called procedure (in field `CALLED BY`) and the calling procedure (in field `CALLS`). At this point we realized that we were getting some errors due to the fact that a reference to a local variable in `FACTOR` was being entered as a call to a procedure of the same name at a lower level. Therefore, it was decided that all names of variables and constants declared should also be stored in the name-table, with the information that it is not a procedure being given by `ISPROC`.

Finally, we not only had the Call Graph, but also the Called By information and the static nesting of procedures in the input program.

#### 4.3.2 Determination of Strongly Connected Components:

Procedures `FINDCOMPS` and `STRONG` (discussed in sec 4.2) were also implemented in the same program, to determine the strongly connected components of the Call Graph already generated. The listing of the program is attached (see Appendix A).

## CHAPTER 5

### APPLICATION TO A RECURSIVE DESCENT COMPILER

The algorithms developed in Chapter 3 were applied to the PASREL compiler. As mentioned in that chapter, an appreciable reduction in the total memory requirement was not obtained.

The results obtained on the application of the program discussed in Section 4.3 to the PASREL compiler are attached. The program gives the strongly connected components, the calls by each procedure (the Call Graph), the procedures which call a procedure, and the static nesting of procedures in the program (see Appendix B).

## CHAPTER 6

### CHANGES IN PASREL TO SUPPORT OVERLAYS

Since, in overlaying, the codes of procedures in a program may be overlaying each other, we would be interested in compiling each procedure (or a pre-planned group of procedures) separately. Separate compilation of procedures is possible in PASREL for those procedures which are declared at level 1, by using the M-option. We are interested in a facility that will allow separate compilation of any procedure in a PASCAL program. For this purposes the scope rules of the language require that all the global user definitions with respect to the procedure of interest be available. To enable this, we have considered the two schemes given below.

(i) All the symbols or identifiers declared in a procedure are written out onto the secondary store, in a separate file. In this way, all the symbols declared within different procedures will be available in different files on the secondary store. When a procedure body is to be compiled, those files which contain the symbols which can be referenced by this procedure (according to the rules of PASCAL) are brought into the main memory. One way of doing this is to include an option in PASREL, say S+, which constructs the global symbol table using the named files.

(ii) Only the procedure bodies should be declared EXTERN, so that the declarations within each procedure are treated as usual, i.e., the symbol table is maintained in the normal way.

## CHAPTER 7

### CONCLUSIONS AND SUGGESTIONS FOR FURTHER WORK

After having developed and tried out a number of algorithms to generate Overlay Trees, we concluded that the problem was too complex, involving too many factors, for a completely automatic Overlay Tree Generator to be possible. The minimal requirements are met by the strongly connected components of the Call Graph, therefore this algorithm was implemented and tried out for the PASREL compiler. The results obtained are very encouraging. With all the calls between the strongly connected components available, it would be quite simple to generate the Overlay Tree manually by duplicating or merging nodes.

This algorithm is an instance of interesting language processing tasks. We must explore the possibilities of such models like Call Graphs for getting other properties of programs.

Though it is a basic requirement for overlaying, the implementation of separate compilation of procedures on DEC system-10 would be a useful addition to the features of the compiler. It would make large programs

like the PASREL compiler, easier to handle for editing etc. At present, a minor alteration in a large program causes the editing of the entire file and the re-compilation of the whole program. With the seperate compilation facility, just one procedure may have to be re-compiled.

Our original aim was to suggest a way to break up large programs so that the PASREL compiler could be loaded and run on the local TDC-316 system. We have made such a suggestion, but the implementation of PASCAL on TDC-316 requires a lot of more work. We hope that some one would take up the challenge and go ahead with the task.

## REFERENCES

1. D.W. Barron, Assemblers and Loaders (3rd edition), Macdonald and Jane's Computer Monographs, (1978).
2. S. Rosen (ed.), Programming systems and Languages, McGraw-Hill Book Company, (1967).
3. C.A.R. Hoare and R.H. Perrot (eds.), Operating Systems Techniques, Academic Press, (1972).
4. H.H. Nageli and R. Schoenberger, Preventing Storage Overflows in High-level Languages, Institut für Informatik, ETH, Zürich, (1980).
5. E.M. Reingold, Jurg Nievergelt and Narsingh Deo, Combinatorial Algorithms - Theory and Practice, Prentice-Hall, Inc., (1977).
6. DEC System-10 Link Reference Manual, (1978).
7. Narsing Deo, Graph Theory with applications to Engineering and Computer Science, Prentice-Hall, Inc., (1974).

APPENDIX A  
-----

PROGRAM LISTING

This program carries out Syntax Analysis,  
generates the Call Graph (alongwith the Called  
by and Static Nesting information), and  
determines its Strongly Connected Components.



const

A - 1

```
NORW = 39;  
AL = 10;  
LMAX = 132;  
NOERRMESS = '**** CONGRATS! YOU WIN !!NO ERRORS DETECTED';  
EOMESS = 'NO INPUT FILE';  
EMES = 'ERRORS DETECTED';  
LEVMAX = 9;  
STMAX = 200;  
NOSP = 15;
```

type

```
SYMBOL =  
(NIL, IDENT, INTNUM, REALNUM, PLUS, MINUS, TIMES, SLASH, POINTER,  
LPAREN, RPAREN, LBRACKET, RBRACKET, EQSYM, NESYM, LTSYM, FOFSYM,  
LESYM, GTSYM, GESYM, ASSIGN, COMMA, PERIOD, SEMICOLON, COLON,  
STRING, ANDSYM, ORSYM, NOTSYM, DIVSYM, MODSYM, BEGINSYM, ENDSYM,  
TESYM, CONSTSYM, PACKEDSYM,  
THENSYM, ELSESYM, WHILESYM, DOSYM, REPEATSYM, UNTILSYM,  
TYPESYM, VARSYM, ARRAYSYM, OFSYM, FILESYM, RECORDSYM,  
FUNCSYM, PROCSYM, PROGSYM, INSYM, FORWARDSYM, SETSYM,  
LOOPSYM, EXITSYM, OTHERSYM, INITPROCSYM, EXTERNSYM,  
FORSYM, TOSYM, DOWNTOSYM, CASESYM, GOTOSYM, WITHSYM);
```

ALPHA = packed array [1..AL] of char;

SYMSET = set of SYMBOL;

```
PROCPtr = ^ PROCNODE;  
LISTOFPROC = ^ NEXTPROC;  
NEXTPROC = record  
    PROC : PROCPtr;  
    NEXT : LISTOFPROC;  
end;
```

```
PROCNODE = record  
    NAME : ALPHA;  
    LLINK, RLINK : PROCPtr;  
    case ISPROC : boolean of  
        true : (DECLPROC : PROCPtr;  
                CALLS, CALLED BY, STRONGCOMP :  
                LISTOFPROC;  
                ONCSTACK : boolean;  
                NODENO, ENCLBLKNO : integer )  
    end;
```

STACK = array [0..STMAX] of PROCPtr;

var

CH : char;

SYM : SYMBOL;

WORD : array [1..NORW] of ALPHA;

WSYM : array [1..NORW] of SYMBOL;

SSYM : array [char] of SYMBOL;

LINE : packed array [1..LMAX] of char;

CC, LL : 0..LMAX;

ERRCOUNT: 1..100;

CONSTBEGSYM, SIMPTYBEGSYM, SELECTSYS, TYPEBEGSYM, TYPDECL,  
DECLBEGSYM, STATBEGSYM, FACBEGSYM: SYMSET;

DISPLAY : array[0..LEVMAX] of record

A - 2

PRCS : PROCPTR;  
CURRENT : PROCPTR  
end;

JOP : 0..LEVMAX;

PNAME : ALPHA;

STDP : array [1..NOSP] of ALPHA;

NEWVARS : boolean;

LMODE : PROCPTR;

STPTR,OPPTR : integer;

PSTACK,DPSIK : STACK;

procedure HALT;  
begin  
end;

procedure ERROR(N:integer);  
const  
ERRMES='ERROR ';  
begin  
WRITELN(OUTPUT,ERRMES,N);  
WRITELN(TTY,ERRMES,N);  
ERRCOUNT:=ERRCOUNT+1  
end;

procedure NEXTCH;

function CAPITAL(CH:char):char;  
begin CAPITAL:=CH;  
if ORD(CH)>140B then  
CAPITAL:=CHR(ORD(CH)-40B)  
end;

begin  
if CC=LL then  
if EOF(INPUT) then HALT  
else  
begin LL:=0; CC:=0;  
OUTPUT^:=  
PUT(OUTPUT);  
while not(EOLN(INPUT)) do  
begin LL:=LL+1;  
LINE[LL]:=INPUT^;  
OUTPUT^:=INPUT^;  
PUT(OUTPUT);  
GET(INPUT)  
end;  
PUTLN(OUTPUT);  
LL:=LL+1;  
LINE[LL]:=' ';  
GET(INPUT)  
end;  
CC:=CC+1;  
CH:=CAPITAL(LINE[CC])  
end;

procedure GETSYM;

var  
I,J,K : integer;  
A : ALPHA;

function LETTER:boolean;  
begin  
if (ORD(CH)>=ORD('A')) and (ORD(CH)<=ORD('Z'))  
LETTER:=true  
else LETTER:=false  
end;

```

function DIGIT:boolean;
begin
  if (ORD(CH)>=ORD('0')) and (ORD(CH)<=ORD('9')) then
    DIGIT:=true
  else DIGIT:=false
  end;
begin
  while CH=' ' do NEXTCH;
  if LETTER then
    begin
      K:=0;
      while DIGIT or
        LETTER do
        begin
          if K<AL then
            begin K:=K+1; A[K]:=CH
          end;
          NEXTCH
        end;
      while K<AL do
        begin K:=K+1; A[K]:=' '
      end;
      I:=1; J:=NORW;
      repeat
        K:=(I+J) div 2;
        if A<=WORD[K] then J:=K-1;
        if A>=WORD[K] then I:=K+1;
      until I>J;
      if I-1>J then SYM:=WSYM[K]
      else
        begin SYM:=IDENT; PNAME := A
        end
      end
    end
  else
    if DIGIT then
      begin
        while DIGIT do NEXTCH;
        SYM:=INTNUM;
        if CH='R' then NEXTCH
        else
          begin
            if CH='.' then
              begin NEXTCH;
              if CH='.' then CH:=':'
            else
              if DIGIT then
                begin SYM:=REALNUM;
                while DIGIT do
                  NEXTCH
                end
              else
                begin
                  SYM:=NUL;
                  ERROR(1);
                  GETSYM
                end
              end;
            if (CH='E') or (CH='e') then
              begin NEXTCH;
              if (CH='+' or (CH='-')) then NEXTCH;
              if DIGIT then
                begin SYM:=REALNUM;
                while DIGIT do
                  NEXTCH
                end
              else
                begin
                  SYM:=NUL;
                  ERROR(2);
                  GETSYM
                end
              end
            end
          end
        end
      end
    end
  end
end
end
end
end

```

```

else
  if CH="" then
    repeat
      NEXTCH;
      while CH<>"" do NEXTCH;
      NEXTCH;
      SYM:=STRING
    until CH<>""
  else
    if CH='<' then
      begin NEXTCH;
      if CH='>' then
        begin SYM:=NESYM; NEXTCH
        end
      else
        if CH='=' then
          begin SYM:=LESYM; NEXTCH
          end
        else SYM:=LTSYM
        end
    else
      if CH='>' then
        begin NEXTCH;
        if CH='=' then
          begin SYM:=GESYM; NEXTCH
          end
        else SYM:=GTSYM
        end
    else
      if CH=':' then
        begin NEXTCH;
        if CH='=' then
          begin SYM:=ASSIGN; NEXTCH
          end
        else SYM:=COLON
        end
    else
      if CH='.' then
        begin NEXTCH;
        if CH='.' then
          begin SYM:=COLON; NEXTCH
          end
        else SYM:=PERIOD
        end
    else
      if CH='(' then
        begin NEXTCH;
        if CH='*' then
          begin NEXTCH;
          repeat
            while CH<>'*' do NEXTCH;
            NEXTCH
          until CH=')';
          SYM:=NUL;
          NEXTCH;
          GETSYM
          end
        else
          SYM:=LPAREN
        end
    else
      if CH='%' then
        begin
          repeat
            NEXTCH
          until CH='\';
          SYM:=NUL;
          NEXTCH;
          GETSYM
        end
      end
    end
  end
end

```

```

if (CH='+') or (CH='-' ) or
(CH='*') or (CH='/' ) or
(CH='^') or (CH='=' ) or (CH='(')
or (CH=')') or
(CH='[') or (CH=']' ) or (CH=',')
or (CH=';') or
(CH='#') then
begin
SYM:=SSYM[CH]; NEXTCH
end
else
begin
SYM:=NUL;
ERROR(3);
NEXTCH;
GETSYM
end;

```

```
end;
```

```

function TESTSYM(LEX:SYMBOL):boolean;
begin TESTSYM := LEX=SYM
end;

```

```

function TESTSYMINSet(LEXSET:SYMSET):boolean;
begin TESTSYMINSet := SYM in LEXSET
end;

```

```

procedure TEST (S1,S2:SYMSET;N:integer);
begin
if not TESTSYMINSet(S1) then
begin ERROR(N); S1 := S1 + S2;
while not TESTSYMINSet(S1) do GETSYM
end
end;

```

```

procedure CHECKSYM(CSYM:SYMBOL;ERR:integer);
begin
if TESTSYM(CSYM) then GETSYM
else ERROR(ERR)
end;

```

```

procedure BSTINSERT(var INPROC : PROCPTR);
var
LPROC,LPROC1 : PROCPTR;
LLEFT,ENTRYDONE : boolean;
begin
ENTRYDONE := false;
LPROC := DISPLAY[ TOP ], PROCS;
if LPROC = nil
then DISPLAY[ TOP ], PROCS := INPROC
else
begin
repeat
LPROC1 := LPROC;
if LPROC ^ . NAME <= INPROC ^ . NAME
then
begin
if LPROC ^ . NAME = INPROC ^ . NAME
then
begin ENTRYDONE := true; INPROC := LPROC
end;
LPROC := LPROC ^ . RLINK; LLEFT := false
end
else
begin
LPROC := LPROC ^ . LLINK; LLEFT := true
end
until LPROC = nil;
if not ENTRYDONE
then
if LLEFT
then LPROC1 ^ . LLINK := INPROC
else LPROC1 ^ . RLINK := INPROC
end
end;

```

```

procedure ENTERVAR;
var
  NEWVAR : PROCPTR;
begin
  NEW( NEWVAR, false );
  with NEWVAR do
    begin
      NAME := PNAME; LLINK := nil; RLINK := nil
    end;
  BSTINSERT( NEWVAR )
end;

procedure SIGNEDCONST(FSYS:SYMSET);
begin
  if ((TESTSYM(PLUS)) or (TESTSYM(MINUS))) then GETSYM;
  if TESTSYM(IDENT) then GETSYM
  else
    if ((TESTSYM(INTNUM)) or (TESTSYM(REALNUM))) then
      GETSYM
    else TEST([1,FSYS,101])
  end;
end;

procedure CONSTANTList( FSYS : SYMSET );
begin
  if TESTSYM( STRING )
  then GETSYM
  else SIGNEDCONST( FSYS + [COLON] );
  if SYM = COMMA
  then
    begin
      GETSYM;
      CONSTANTList( FSYS )
    end
  end;
end;

procedure CONSTDEF(FSYS:SYMSET);
begin
  TEST(CONSTBEGSYM,FSYS,102);
  if TESTSYM(STRING) then GETSYM
  else SIGNEDCONST(FSYS)
end;

procedure CONSTDECL(FSYS:SYMSET);
begin
  TEST([IDENT1,FSYS,103);
  while TESTSYM(IDENT) do
    begin ENTERVAR; GETSYM;
      if TESTSYM(EQSYM) then
        begin GETSYM;
          CONSTDEF(FSYS+[SEMICOLON]);
          CHECKSYM(SEMICOLON,5)
        end
      else ERROR(4)
    end;
  TEST([IDENT1+FSYS,[1,104)
end;

procedure IDENTLIST(FSYS:SYMSET);
begin
  TEST([IDENT1,FSYS,606);
  if TESTSYM(IDENT) then
    begin
      if NEWVARS then ENTERVAR;
      GETSYM
    end;
  while TESTSYM(COMMA) do
    begin GETSYM;
      IDENTLIST(FSYS+[COMMA])
    end
  end;
end;

```

```

20 procedure SIMPLETYPE(FSYS:SYMSET);
30 begin
40   TEST(SIMPTYPEGSym,FSYS,110);
50   if TESTSYM(INSet(SIMPTYPEGSym) then
60     begin
70       if TESTSYM(STRING) then
80         begin GETSYM;
90           CHECKSYM(COLON,20); CHECKSYM(STRING,21)
00         end
10       else
20         if TESTSYM(LPAREN) then
30           begin GETSYM; NEWVARS := true;
40             IDENTLIST(FSYS+[RPAREN]);
50             NEWVARS := false;
60             CHECKSYM(RPAREN,22)
70           end
80         else
90           begin SIGNEDCONST(FSYS+[COLON]);
00             if TESTSYM(COLON) then
10               begin GETSYM; SIGNEDCONST(FSYS)
20               end
30             end
40           end
50         end;
60       end;
70     end;
80   end;
90   end;

```

```

70 procedure TYPEDEF(FSYS:SYMSET);
80 forward;
90

```

```

10 procedure ARRAYTYPE(FSYS:SYMSET);
20 begin TEST([LBRACKET],FSYS,23);
30 if TESTSYM(LBRACKET) then
40   begin GETSYM;
50     SIMPLETYPE(FSYS+[COMMA,RBRACKET]);
60     while TESTSYM(COMMA) do
70       begin SIMPLETYPE(FSYS+[COMMA,RBRACKET])
80       end;
90     if TESTSYM(RBRACKET) then GETSYM
00     else ERROR(24);
10     if TESTSYM(OFSYM) then GETSYM
20     else ERROR(25);
30     TYPEDEF(FSYS)
40   end
50 end;
60 end;
70

```

```

80 procedure FIELDLIST(FSYS:SYMSET);
90 begin
00   while TESTSYM( IDENT ) do
10     begin
20       IDENTLIST( FSYS + [COLON] );
30       CHECKSYM( COLON,951 );
40       TYPEDEF( FSYS + [SEMICOLON,ENDSYM,CASESYM] );
50       if TESTSYM( SEMICOLON ) then GETSYM
60       end;
70       if TESTSYM( CASESYM ) then
80         begin
90           GETSYM;
00           CHECKSYM( IDENT,952 );
10           if TESTSYM( COLON ) then
20             begin GETSYM; CHECKSYM( IDENT,953 )
30             end;
40           CHECKSYM( OFSYM,954 );
50           loop
60             CONSTANTList( FSYS + [COLON] );
70             CHECKSYM( COLON,955 );
80             CHECKSYM( LPAREN,956 );
90             FIELDLIST( FSYS + [RPAREN] );
00             CHECKSYM( RPAREN,957 );
10           exit if SYM # SEMICOLON;
20           GETSYM;
30         end
40       end
50     end;
60   end;
70 end;
80

```

```

procedure RECCARR(FSYS:SYMSET);
begin
  if TESTSYM(RECORDSYM) then
    begin GETSYM;
      FIELDLIST(FSYS+[ENDSYM]);
      CHECKSYM(ENDSYM,28)
    end
  else
    begin CHECKSYM(ARRAYSYM,29);
      ARRAYTYPE(FSYS)
    end
  end;
end;

```

```

procedure SETTYPE(FSYS:SYMSET);
begin CHECKSYM(OFSYM,30);
  SIMPLETYPE(FSYS)
end;

```

```

procedure TYPEDEF;
begin
  TEST(TYPEBEGSYM,FSYS,112);
  if TESTSYMINSet(TYPEBEGSYM) then
    begin
      if TESTSYM(SETSYM) then
        begin GETSYM; SETTYPE(FSYS)
      end
    else
      if TESTSYM(FILESYM) then
        begin GETSYM;CHECKSYM( OFSYM,930 );
          TYPEDEF( FSYS )
        end
      else
        if TESTSYM(PACKEDSYM) then
          begin GETSYM; TYPEDEF( FSYS )
        end
      else
        if TESTSYM(POINTER) then
          begin GETSYM; CHECKSYM(IDENT,31)
        end
      else
        if (TESTSYMINSet( [RECORDSYM,ARRAYSYM] ))
          then RECARR(FSYS)
        else SIMPLETYPE(FSYS);
    TEST(FSYS,[1],113)
  end
end;

```

```

procedure TYPEDECL(FSYS:SYMSET);
begin
  TEST([IDENT],FSYS,103);
  while TESTSYM(IDENT) do
    begin ENTERVAR; GETSYM;
      CHECKSYM(EOSYM,4);
      TYPEDEF(FSYS+[SEMICOLON]);
      CHECKSYM(SEMICOLON,5)
    end;
  TEST([IDENT]+FSYS,[1],104)
end;

```

```

procedure VARDECL(FSYS:SYMSET);
begin
  NEWVARS := true;
  repeat
    IDENTLIST(FSYS+[COLON]);
    CHECKSYM(COLON,32);
    TYPEDEF(FSYS+[SEMICOLON]);
    CHECKSYM(SEMICOLON,33)
  until (not TESTSYM(IDENT)) and not TESTSYMINSet(TYPDECL);
  NEWVARS := false
end;

```



```

10 procedure ENTERPROC;
11 var
12   NEWPROC : PROCPTR;
13 begin
14   NEW( NEWPROC, true );
15   with NEWPROC ^ do
16     begin
17       NAME := PNAME; LLINK := nil; RLINK := nil;
18       CALLED BY := nil; ENCLBLKNO := 0; NODENO := 0;
19       DECLPROC := nil; CALLS := nil; STRONGCOMP := nil;
20       ONCSTACK := false
21     end;
22   BSTINSERT(NEWPROC);
23   TOP := TOP + 1;
24   DISPLAY( TOP ). PROCS := nil;
25   DISPLAY( TOP ). CURRENT := NEWPROC
26 end;

```

```

10 procedure ENTERSTDPROCS;
11 var
12   T : integer;
13   STDPROC : PROCPTR;
14 begin
15   T := 1;
16   DISPLAY( TOP ). PROCS := nil;
17   repeat
18     NEW( STDPROC, true );
19     with STDPROC ^ do
20       begin
21         NAME := STDPRI1; LLINK := nil; RLINK := nil;
22         CALLED BY := nil; DECLPROC := nil; ENCLBLKNO := 0;
23         CALLS := nil; STRONGCOMP := nil; NODENO := 0;
24         ONCSTACK := false
25       end;
26     if T = 1 then DISPLAY( 1 ). CURRENT := STDPROC;
27     BSTINSERT(STDPROC);
28     T := T + 1
29   until T > 14
30 end;

```

```

10 function SEARCHPROC(SNAME : ALPHA ) : PROCPTR;
11 var
12   LPROC : PROCPTR;
13   LTOP : integer;
14   FLAG : boolean;
15 begin
16   LTOP := TOP;
17   repeat
18     LPROC := DISPLAY( LTOP ). PROCS;
19     FLAG := LPROC # nil;
20     while FLAG do
21       begin
22         if LPROC ^ . NAME < SNAME
23           then LPROC := LPROC ^ . RLINK
24         else
25           if LPROC ^ . NAME = SNAME
26             then FLAG := false
27           else LPROC := LPROC ^ . LLINK;
28         if FLAG then FLAG := LPROC # nil
29       end;
30     LTOP := LTOP - 1;
31     if LPROC # nil then FLAG := LPROC ^ . NAME = SNAME
32   until (LTOP < 0) or FLAG;
33   if FLAG then FLAG := LPROC ^ . ISPROC;
34   if FLAG then SEARCHPROC := LPROC
35   else SEARCHPROC := nil
36 end;

```

```

procedure ENTERCALL;
var
  LLP : LISTOFPROC;
  FLAG : boolean;
  CALLEDPROC : PROCPTR;
begin
  LLP := DISPLAY[ TOP 1, CURRENT ^, CALLS;
  FLAG := LLP # nil;
  while FLAG do
    begin
      if LLP ^, PROC ^, NAME = PNAME then FLAG := false;
      if FLAG then
        begin LLP := LLP ^, NEXT; FLAG := LLP # nil
        end
      end;
    if LLP = nil
    then
      begin
        CALLEDPROC := SEARCHPROC( PNAME );
        if CALLEDPROC # nil (* CALLEDPROC=NIL INDICATES IT IS A
                           VARIABLE*)
        then
          begin
            NEW( LLP );
            LLP ^, PROC := CALLEDPROC;
            LLP ^, NEXT := DISPLAY[ TOP 1, CURRENT ^, CALLS;
            DISPLAY[ TOP 1, CURRENT ^, CALLS := LLP;
            NEW( LLP );
            LLP ^, PROC := DISPLAY[ TOP 1, CURRENT;
            LLP ^, NEXT := CALLEDPROC ^, CALLED BY;
            CALLEDPROC ^, CALLED BY := LLP
          end
        end
      end;
    end;
end;

procedure EXPRESSION(FSYS:SYMSET);
forward;

procedure EXPLIST(FSYS:SYMSET);
begin
  EXPRESSION(FSYS+[COMMA,COLON]);
  if TESTSYMINSet( [COMMA,COLON] ) then
    begin GETSYM;
    EXPLIST(FSYS)
    end;
  TEST(FSYS,[1],601)
end;

procedure SELECTOR(FSYS:SYMSET);
begin
  if TESTSYMINSet(SELECTSYS) then
    begin
      if TESTSYM(LBRACKET) then
        begin GETSYM;
        EXPLIST(FSYS+[RBRACKET]);
        CHECKSYM(RBRACKET,9)
        end
      else
        if TESTSYM(PERIOD) then
          begin GETSYM;
          CHECKSYM(IDENT,10)
          end
        else
          if TESTSYM(POINTER) then GETSYM;
          SELECTOR(FSYS)
        end
      end;
    end;

procedure FUNORVAR(FSYS:SYMSET);
begin SELECTOR(FSYS+[LPAREN]);
  if TESTSYM(LPAREN) then
    begin GETSYM;
    EXPLIST(FSYS+[RPAREN]);
    CHECKSYM(RPAREN,11)
    end
  end;
end;

```

```

30 procedure FACTOR(FSYS:SYMSET);
40 begin
50   TEST(FACBEGSYM,FSYS,107);
60   if TESTSYMINSet(FACBEGSYM) then
70     begin
80       if TESTSYM(IDENT) then
90         begin ENTERCALL; GETSYM; FUNDRVAR(FSYS)
100        end
110       else
120         if (TESTSYMINSet( [INTNUM,REALNUM,STRING])) then
130           GETSYM
140         else
150           if TESTSYM(NTSYM) then
160             begin GETSYM; FACTOR(FSYS)
170             end
180           else
190             if TESTSYM(LPAREN) then
200               begin GETSYM;
210               EXPRESSION(FSYS+[RPAREN]);
220               CHECKSYM( RPAREN,12)
230             end
240           else
250             if TESTSYM(LBRACKET) then
260               begin
270                 GETSYM;
280                 if not (TESTSYM(RBRACKET)) then
290                   EXPLIST(FSYS+[RBRACKET]);
300                 CHECKSYM(RBRACKET,13)
310               end
320             end;
330           TEST(FSYS,[1,108)
340         end;

```

```

350 procedure TERM(FSYS:SYMSET);
360 begin FACTOR(FSYS+[TIMES,SLASH,DIVSYM,MODSYM,ANDSYM]);
370 while (TESTSYMINSet( [TIMES,SLASH,DIVSYM,MODSYM,ANDSYM]))
380 do
390   begin GETSYM; FACTOR(FSYS+[TIMES,SLASH,DIVSYM,MODSYM,
400   ANDSYM])
410   end
420 end;

```

```

430 procedure SIMEXP(FSYS:SYMSET);
440 begin
450   if (TESTSYMINSet( [PLUS,MINUS])) then GETSYM;
460   TERM(FSYS+[PLUS,MINUS,ORSYM]);
470   while (TESTSYMINSet( [PLUS,MINUS,ORSYM])) do
480     begin GETSYM; TERM(FSYS+[PLUS,MINUS,ORSYM])
490     end
500   end;

```

```

510 procedure EXPRESSION;
520 begin SIMEXP(FSYS+[EQSYM,NESYM,LTSYM,LESYM,GTSYM,GESYM,INSYM]
530 );
540   if (TESTSYMINSet( [EQSYM,NESYM,LTSYM,LESYM,GTSYM,GESYM,
550   INSYM])) then
560     begin GETSYM;
570     SIMEXP(FSYS)
580     end
590   end;

```

```

600 procedure PUSH(var STNODE : PROCPTR;var STK : STACK ; var PTR :
610   integer);
620 begin
630   PTR := PTR + 1;
640   if PTR > STMAX
650     then ERROR( 300 )
660   else STK[PTR] := STNODE
670   end;

```

```

10 procedure POP(var STNODE : PROCPTR; var STK : STACK ; var PTR :
11     integer);
12 begin
13     if PTR = 0
14     then ERROR( 301 )
15     else
16     begin STNODE := STK[PTR]; PTR := PTR - 1
17     end
18 end;
19
20 function STACKEMPTY(PTR : integer) : boolean;
21 begin
22     if PTR = 0 then STACKEMPTY := true
23     else STACKEMPTY := false
24 end;
25
26 procedure FINDCOMPS;
27 var
28     COMPSTK : STACK;
29     CPTR, I : integer;
30
31 function MIN( L, M : integer ) : integer;
32 begin
33     if L < M then MIN := L
34     else MIN := M
35 end;
36
37 procedure ENTERCOMP( var LPROC, LLPROC : PROCPTR );
38 var
39     COMP : LISTOFPROC;
40 begin
41     NEW( COMP );
42     COMP^.PROC := LPROC;
43     COMP^.NEXT := LPROC^.STRONGCOMP;
44     LPROC^.STRONGCOMP := COMP
45 end;
46
47 procedure STRONG( var LPROC : PROCPTR );
48 var
49     LLPROC : PROCPTR;
50     LLP : LISTOFPROC;
51     FLAG : boolean;
52 begin
53     I := I + 1;
54     LPROC^.NODENO := I;
55     LPROC^.ENCLBLKNO := I;
56     PUSH( LPROC, COMPSTK, CPTR );
57     LPROC^.ONCSTACK := true;
58     LLP := LPROC^.CALLS;
59     while LLP # nil do
60     begin
61         LLPROC := LLP^.PROC;
62         if LLPROC^.NODENO = 0 then
63             begin STRONG( LLPROC );
64                 LPROC^.ENCLBLKNO := MIN( LPROC^.ENCLBLKNO,
65                     LLPROC^.ENCLBLKNO )
66             end
67         else
68             if LLPROC^.NODENO < LPROC^.NODENO then
69                 if LLPROC^.ONCSTACK then
70                     LPROC^.ENCLBLKNO := MIN( LPROC^.ENCLBLKNO,
71                         LLPROC^.NODENO );
72                 LLP := LLP^.NEXT
73             end;
74         if LPROC^.ENCLBLKNO = LPROC^.NODENO then
75             begin
76                 POP( LLPROC, COMPSTK, CPTR );
77                 LLPROC^.ONCSTACK := false;
78                 FLAG := LLPROC^.NODENO >= LPROC^.NODENO;
79                 while FLAG do
80                 begin
81                     ENTERCOMP( LPROC, LLPROC );
82                     if STACKEMPTY( CPTR )
83                     then FLAG := false
84                     else

```

```

        POP( LLPROC, COMPSTK, CPTR );
        LLPROC ^ . ONCSTACK := false;
        FLAG := LLPROC ^ . NODENO >= LPROC ^ . NODENO
    end;
end;
if LLPROC ^ . NODENO < LPROC ^ . NODENO then
    begin PUSH( LLPROC, COMPSTK, CPTR );
        LLPROC ^ . ONCSTACK := true
    end
end
end;
end;

procedure TRAVERSETree;
begin
    while LNODE # nil do
        begin PUSH( LNODE, PSTACK, STPTR ); LNODE := LNODE ^ .
            LLINK
        end;
        while not STACKEMPTY( STPTR ) do
            begin
                POP( LNODE, PSTACK, STPTR );
                if LNODE ^ . ISPROC then
                    begin
                        if LNODE ^ . NODENO = 0 then STRONG( LNODE );
                        if LNODE ^ . DECLPROC # nil then PUSH( LNODE, DPSTK,
                            DPPTR )
                        end;
                        LNODE := LNODE ^ . RLINK;
                    end
                while LNODE # nil do
                    begin PUSH( LNODE, PSTACK, STPTR ); LNODE := LNODE ^ .
                        LLINK
                    end
                end
            end;
        end;
    begin % FINDCOMPS \
        STPTR := 0; DPPTR := 0; CPTR := 0; I := 0;
        LNODE := DISPLAY( 0 1, PROCS;
        STRONG( LNODE );
        loop
            LNODE := LNODE ^ . DECLPROC;
            TRAVERSETree;
        exit if STACKEMPTY( DPSTK );
        POP( LNODE, DPSTK, DPPTR )
    end
end;
end;

```

procedure WRITEPINFO;

procedure WRITEPTREE;

procedure WRITENODE;

```

    var
        COUNT : integer;
    begin
        if ( LNODE ^ . CALLS = nil ) and ( LNODE ^ . CALLED BY =
            nil )
        then
            begin
                WRITELN;
                WRITELN( LNODE ^ . NAME, ' is not called in this program.'
                )
            end
        else
            begin
                WRITELN;
                WRITELN( LNODE ^ . NAME );
                if LNODE ^ . STRONGCOMP # nil then
                    begin
                        WRITELN( ' It is the root of a strongly connected componen
                        );
                        WRITELN( ' which consists of the following procedure(s)
                        );
                    end
                end
            end
        end
    end
end;

```

```

17
while LNODE ^ . STRONGCOMP # nil do
begin
WRITE( ' ', LNODE ^ . STRONGCOMP ^ . PROC ^ .
NAME );
LNODE ^ . STRONGCOMP := LNODE ^ . STRONGCOMP ^
NEXT;
COUNT := COUNT + 1;
if COUNT > 5 then
begin WRITELN; COUNT := 1
end
end;
WRITELN
end;
if LNODE ^ . CALLS # nil
then
begin
WRITELN( ' It calls the following procedures : '
);
COUNT := 1;
while LNODE ^ . CALLS # nil do
begin
WRITE( ' ', LNODE ^ . CALLS ^ . PROC ^ . NAME
);
LNODE ^ . CALLS := LNODE ^ . CALLS ^ . NEXT;
COUNT := COUNT + 1;
if COUNT > 5 then
begin WRITELN; COUNT := 1
end
end;
WRITELN
end;
if LNODE ^ . CALLED BY # nil
then
begin
WRITELN( ' It is called by the following procedures : '
);
COUNT := 1;
while LNODE ^ . CALLED BY # nil do
begin
WRITE( ' ', LNODE ^ . CALLED BY ^ . PROC ^ .
NAME );
LNODE ^ . CALLED BY := LNODE ^ . CALLED BY ^ .
NEXT;
COUNT := COUNT + 1;
if COUNT > 5 then
begin WRITELN; COUNT := 1
end
end;
WRITELN
end
end
end;
begin (* WRITEPTREE *)
while LNODE # nil do
begin PUSH(LNODE, PSTACK, STPTR); LNODE := LNODE ^ .
LLINK
end;
while not STACKEMPTY(STPTR) do
begin
POP(LNODE, PSTACK, STPTR);
if LNODE ^ . ISPROC then
begin WRITENODE;
if LNODE ^ . DECLPROC # nil then PUSH(LNODE, DPSTK,
DPPTR)
end;
LNODE := LNODE ^ . RLINK;
while LNODE # nil do
begin PUSH(LNODE, PSTACK, STPTR); LNODE := LNODE ^ .
LLINK
end
end
end;
end;

```

```

begin (* WRITEPINFO *)
PAGE;
WRITELN;
WRITELN(' LIST OF PROCEDURES AND FUNCTIONS IN THE PROGRAM');
WRITELN('): -----');
WRITELN('): with static nesting and calls information');
STDTR := 0; DPPTR := 0;
LNODE := DISPLAY( 0 1, PROCS);
WRITEPTREE;
while not STACKEMPTY(DPPTR) do
begin
POP(LNODE, DPSTK, DPPTR);
WRITELN;
WRITELN(' Following procedures are declared in ', LNODE);
WRITELN(' .NAME); -----');
LNODE := LNODE ^ . DECLPROC;
WRITEPTREE
end
end;

procedure STATEMENT(FSYS:SYMSET);
forward;

procedure STATLIST(FSYS:SYMSET);
begin
STATEMENT(FSYS+[SEMICOLON]);
if TESTSYM(SEMICOLON) then
begin GETSYM;
STATLIST(FSYS)
end;
TEST(FSYS, [1, 600])
end;

procedure IFSTAT(FSYS:SYMSET);
begin EXPRESSION(FSYS+[THENSYM]);
CHECKSYM(THENSYM, 14);
STATEMENT(FSYS+[ELSESYM]);
if TESTSYM(ELSESYM) then
begin GETSYM; STATEMENT(FSYS)
end
end;

procedure WHILESTAT(FSYS:SYMSET);
begin EXPRESSION(FSYS+[DOSYM]);
CHECKSYM(DOSYM, 15);
STATEMENT(FSYS)
end;

procedure REPEATSTAT(FSYS:SYMSET);
begin STATLIST(FSYS+[UNTILSYM]);
CHECKSYM(UNTILSYM, 16);
EXPRESSION(FSYS)
end;

procedure OTHERSTAT(FSYS:SYMSET);
begin SELECTOR(FSYS+[ASSIGN]);
if TESTSYM(ASSIGN) then
begin GETSYM; EXPRESSION(FSYS)
end
else
begin
ENTERCALL;
if TESTSYM(LPAREN) then
begin GETSYM;
EXPLIST(FSYS+[RPAREN]);
CHECKSYM(RPAREN, 17)
end
end
end;
end;

```

```

procedure FORSTAT( FSYS : SYMSET );
begin
  CHECKSYM( IDENT,601 );
  CHECKSYM( ASSIGN,602 );
  EXPRESSION( FSYS + [ITOSYM,DOWNTOSYM] );
  if TESTSYMINSet( [ITOSYM,DOWNTOSYM] )
  then GETSYM
  else ERROR( 603 );
  EXPRESSION( FSYS + [DOSYM] );
  CHECKSYM( DOSYM,603 );
  STATEMENT( FSYS );
end;

procedure WITHSTAT( FSYS : SYMSET );
begin
  repeat
    if TESTSYM( COMMA ) then GETSYM;
    CHECKSYM( IDENT,605 );
    SELECTOR( FSYS + [COMMA,DOSYM] );
  until SYM # COMMA;
  CHECKSYM( DOSYM,606 );
  STATEMENT( FSYS );
end;

procedure CASESTAT( FSYS : SYMSET );
begin EXPRESSION( FSYS + [OFSYM] );
  if not TESTSYM( OFSYM ) then ERROR( 607 )
  else
    repeat
      GETSYM;
      if TESTSYM( OTHERSYM ) then GETSYM
      else CONSTANTLIST( FSYS + [COLON] );
      CHECKSYM( COLON,608 );
      STATEMENT( FSYS + [ENDSYM,SEMICOLON] );
    until SYM # SEMICOLON;
    if TESTSYM( ENDSYM )
    then GETSYM
    else TEST( [I,FSYS,609 ]
  end;

procedure GOTOSTAT( FSYS : SYMSET );
begin
  if TESTSYM( INTNUM )
  then GETSYM
  else ERROR( 610 )
end;

procedure LABELSTAT( FSYS : SYMSET );
begin
  CHECKSYM( COLON,611 );
  STATEMENT( FSYS );
end;

procedure LOOPSTAT( FSYS : SYMSET );
begin
  STATLIST( FSYS + [EXITSYM] );
  CHECKSYM( EXITSYM,612 );
  CHECKSYM( IFSYM,613 );
  EXPRESSION( FSYS + [SEMICOLON,ENDSYM] );
  if TESTSYM( SEMICOLON ) then
    begin GETSYM; STATLIST( FSYS + [ENDSYM] )
    end;
  CHECKSYM( ENDSYM,615 )
end;

procedure STATEMENT;
begin
  TEST(FSYS+[IDENT],FSYS,109);
  if TESTSYMINSet(STATBEGSYM+[IDENT]) then
    begin
      if TESTSYM(BEGINSYM) then
        begin GETSYM; STATLIST(FSYS+[ENDSYM]);
          CHECKSYM(ENDSYM,18)
        end
    end

```



```

if TESTSYM(IFSVM) then
  begin GETSYM; IFSTAT(FSYS)
end
else
  if TESTSYM(WHILESYM) then
    begin GETSYM; WHILESTAT(FSYS)
  end
  else
    if TESTSYM(REPEATSYM) then
      begin GETSYM; REPEATSTAT(FSYS)
    end
    else
      if TESTSYM(IDENT) then
        begin GETSYM;
          OTHERSTAT(FSYS)
        end
      else
        case SYM of
          LOOPSYM :
            begin GETSYM; LOOPSTAT( FSYS )
          end;
          WITHSYM :
            begin GETSYM; WITHSTAT( FSYS )
          end;
          CASESYM :
            begin GETSYM; CASESTAT( FSYS )
          end;
          FORSYM :
            begin GETSYM; FORSTAT( FSYS )
          end;
          GOTOSYM :
            begin GETSYM; GOTOSTAT( FSYS )
          end;
          INTNUM :
            begin GETSYM; LABELSTAT( FSYS )
          end
        end
      end
    end
  end;
end;

```

```

procedure PARAMETIDENT1st(FSYS:SYMSET);
begin TEST([IDENT1,FSYS,121);
  if TESTSYM(IDENT) then
    begin ENTERVAR; GETSYM;
    while TESTSYM(COMMA) do
      begin GETSYM;
        if TESTSYM(IDENT) then
          begin ENTERVAR;GETSYM
        end
        else ERROR(42)
      end
    end;
  TEST(FSYS,[1,122)
end;

```

```

procedure LISTOFFPARAMets(FSYS:SYMSET);
begin
  if (TESTSYMINSet( [VARSYM,IDENT])) then
    begin
      if TESTSYM(VARSYM) then
        begin GETSYM;
          PARAMETIDENT1st(FSYS+[COLON,IDENT1]);
          CHECKSYM(COLON,52);
          CHECKSYM(IDENT,53)
        end
      else
        begin
          PARAMETIDENT1st(FSYS+[COLON,IDENT1]);
          CHECKSYM(COLON,54);
          CHECKSYM(IDENT,55)
        end
      end;
    end;
  end;
end;

```

```

begin GETSYM;
  LISTOFPARAMets(FSYS+[VARSYM,IDENT])
end
end;

procedure PARAMETERList(FSYS:SYMSET);
begin
  if TESTSYM(LPAREN) then
    begin GETSYM;
      LISTOFPARAMets(FSYS+[RPAREN]);
      CHECKSYM(RPAREN,56)
    end
  end;
end;

procedure PROCHEADER(FSYS:SYMSET);
begin CHECKSYM(IDENT,57);
  PARAMETERList(FSYS)
end;

procedure FUNCHEADER(FSYS:SYMSET);
begin CHECKSYM(IDENT,58);
  PARAMETERList(FSYS+[COLON,IDENT]);
  CHECKSYM(COLON,59);
  CHECKSYM(IDENT,60);
  TEST(FSYS,[1,118])
end;

procedure BLOCK(FSYS:SYMSET);
forward;

procedure FUNORPROCDecl(FSYS:SYMSET);
begin
  TEST(FSYS,[1,119]);
  if (TESTSYMINSet([PROCSYM,FUNCSYM])) then
    begin
      if TESTSYM(PROCSYM) then
        begin GETSYM;
          ENTERPROC;
          PROCHEADER(FSYS+[SEMICOLON])
        end
      else
        begin GETSYM;
          ENTERPROC;
          FUNCHEADER(FSYS+[SEMICOLON])
        end;
      CHECKSYM(SEMICOLON,39);
      if TESTSYMINSet([EXTERNSYM,FORWARDSYM]) then
        begin TOP:=TOP-1;GETSYM
        end
      else BLOCK(FSYS+[SEMICOLON]);
      CHECKSYM(SEMICOLON,50);
      FUNORPROCDecl(FSYS+[FUNCSYM,PROCSYM])
    end
  end;
end;

procedure INITPROCS( FSYS : SYMSET );
begin
  while TESTSYM( INITPROCSym ) do
    begin
      GETSYM;
      CHECKSYM( SEMICOLON,900 );
      CHECKSYM( BEGINSYM,901 );
      repeat
        CHECKSYM( IDENT,902 );
        OTHERSTAT( FSYS + [SEMICOLON,ENDSYM] );
        if TESTSYM( SEMICOLON ) then GETSYM
      until SYM # IDENT;
      CHECKSYM( ENDSYM,903 );
      CHECKSYM( SEMICOLON,904 )
    end
  end;
end;

```

```

procedure BLOCK;
begin
  repeat
    if TESTSYM(CONSTSYM) then
      begin GETSYM; CONSTDECL(FSYS)
      end;
    if TESTSYM(TYPESYM) then
      begin GETSYM; TYPEDECL(FSYS)
      end;
    if TESTSYM(VARSYM) then
      begin GETSYM; VARDECL(FSYS)
      end;
    if TESTSYM( INITPROCSYM ) then
      INITPROCS( FSYS );
    if TESTSYM( INSet( [PROCSYM, FUNCSYM] ) then
      FUNDRPROCDECL( FSYS+[BEGINSYM] );
    TEST( [BEGINSYM], DECLBEGSYM+STATBEGSYM, 603 );
  until TESTSYM( INSet( STATBEGSYM );
  CHECKSYM( BEGINSYM, 40 );
  STATLIST( FSYS+[ENDSYM] );
  CHECKSYM( ENDSYM, 41 );
  DISPLAY( TOP1.CURRENT ^ .DECLPROC := DISPLAY( TOP1.PROCS;
  TOP := TOP - 1;
  TEST( [SEMICOLON, PERIOD], FSYS, 120 )
end;

```

```

procedure FILELIST(FSYS:SYMSET);
begin TEST( [IDENT], FSYS, 121 );
  if TESTSYM( IDENT ) then
    begin GETSYM;
      while TESTSYM( COMMA ) do
        begin GETSYM;
          if TESTSYM( IDENT ) then GETSYM
          else ERROR( 42 )
        end
      end;
    TEST( FSYS, [ ], 122 )
  end;

```

```

procedure PROGRAMHEAD(FSYS:SYMSET);
begin
  TEST( [PROGSYM], FSYS, 123 );
  if TESTSYM( PROGSYM ) then
    begin GETSYM;
      if TESTSYM( IDENT ) then
        begin GETSYM;
          if TESTSYM( LPAREN ) then
            begin GETSYM;
              FILELIST( FSYS+[RPAREN] );
              if TESTSYM( RPAREN ) then
                begin GETSYM;
                  if TESTSYM( SEMICOLON ) then GETSYM
                  else ERROR( 43 )
                end
              else ERROR( 46 )
            end
          else ERROR( 45 )
        end
      else ERROR( 44 )
    end;
  TEST( FSYS, [ ], 124 )
end;

```

```

begin
  (* INITIALIZATIONS *)
  WORD[ 1 ] := 'AND'      ;
  WORD[ 2 ] := 'ARRAY'    ;
  WORD[ 3 ] := 'BEGIN'    ;
  WORD[ 4 ] := 'CASE'     ;
  WORD[ 5 ] := 'CONST'    ;
  WORD[ 6 ] := 'DIV'      ;

```

WORD[91] := 'ELSE';  
 WORD[101] := 'END';  
 WORD[111] := 'EXIT';  
 WORD[121] := 'EXTERN';  
 WORD[131] := 'FILE';  
 WORD[141] := 'FOR';  
 WORD[151] := 'FORWARD';  
 WORD[161] := 'FUNCTION';  
 WORD[171] := 'GOTO';  
 WORD[181] := 'IF';  
 WORD[191] := 'IN';  
 WORD[201] := 'INITPROCED';  
 WORD[211] := 'LOOP';  
 WORD[221] := 'MOD';  
 WORD[231] := 'NOT';  
 WORD[241] := 'OF';  
 WORD[251] := 'OR';  
 WORD[261] := 'OTHERS';  
 WORD[271] := 'PACKED';  
 WORD[281] := 'PROCEDURE';  
 WORD[291] := 'PROGRAM';  
 WORD[301] := 'RECORD';  
 WORD[311] := 'REPEAT';  
 WORD[321] := 'SET';  
 WORD[331] := 'THEN';  
 WORD[341] := 'TO';  
 WORD[351] := 'TYPE';  
 WORD[361] := 'UNTIL';  
 WORD[371] := 'VAR';  
 WORD[381] := 'WHILE';  
 WORD[391] := 'WITH';

WSYM[11] := ANDSYM;  
 WSYM[21] := ARRAYSYM;  
 WSYM[31] := BEGINSYM;  
 WSYM[41] := CASESYM;  
 WSYM[51] := CONSTSYM;  
 WSYM[61] := DIVSYM;  
 WSYM[71] := DDSYM;  
 WSYM[81] := DOWNTOSYM;  
 WSYM[91] := ELSEFSYM;  
 WSYM[101] := ENDSYM;  
 WSYM[111] := EXITSYM;  
 WSYM[121] := EXTERNSYM;  
 WSYM[131] := FILESYM;  
 WSYM[141] := FORSYM;  
 WSYM[151] := FORWARDSYM;  
 WSYM[161] := FUNCSYM;  
 WSYM[171] := GOTOSYM;  
 WSYM[181] := IFSYM;  
 WSYM[191] := INSYM;  
 WSYM[201] := INITPROCSYM;  
 WSYM[211] := LOOPSYM;  
 WSYM[221] := MODSYM;  
 WSYM[231] := NOTSYM;  
 WSYM[241] := OFSYM;  
 WSYM[251] := ORSYM;  
 WSYM[261] := OTHERSYM;  
 WSYM[271] := PACKEDSYM;  
 WSYM[281] := PROCSYM;  
 WSYM[291] := PROGSYM;  
 WSYM[301] := RECORDSYM;  
 WSYM[311] := REPEATSYM;  
 WSYM[321] := SETSYM;  
 WSYM[331] := THENSYM;  
 WSYM[341] := TOSYM;  
 WSYM[351] := TYPESYM;  
 WSYM[361] := UNTILSYM;  
 WSYM[371] := VARSYM;  
 WSYM[381] := WHILESYM;  
 WSYM[391] := WITHSYM;

```

SSYM['+']:=PLUS;
SSYM['-']:=MINUS;
SSYM['*']:=TIMES;
SSYM['/']:=SLASH;
SSYM['^']:=POINTER;
SSYM['=']:=EOSYM;
SSYM['(']:=LPAREN;
SSYM[')']:=RPAREN;
SSYM['[']:=LBRACKET;
SSYM[']']:=RBRACKET;
SSYM[':']:=COMMA;
SSYM[';']:=SEMICOLON;
SSYM['"']:=MESYM;

```

```

STDPR[1] := 'MAINBODY.';
STDPR[2] := 'READ';
STDPR[3] := 'READLN';
STDPR[4] := 'WRITE';
STDPR[5] := 'WRITELN';
STDPR[6] := 'GET';
STDPR[7] := 'PUT';
STDPR[8] := 'RESET';
STDPR[9] := 'REWRITE';
STDPR[10] := 'NEW';
STDPR[11] := 'PACK';
STDPR[12] := 'UNPACK';
STDPR[13] := 'BREAK';
STDPR[14] := 'PAGE';

```

```

CH:=' '; CC:=0; LL:=0;
NEWVARS := false;

```

```

DECLBEGSYM:=[CONSTSYM, VARSYM, TYPESYM, PROCSYM, FUNCSYM,
FORWARDSYM, EXTERNSYM];
STATBEGSYM:=[BEGINSYM, IFSYM, WHILESYM, REPEATSYM, FORSYM, WITHSYM,
CASESYM, GOTOSYM, INTNUM, LOOPSYM];
FACBEGSYM:=[LPAREN, NOTSYM, INTNUM, REALNUM, IDENT, STRING,
LBRACKET];
CONSTBEGSYM:=[PLUS, MINUS, INTNUM, REALNUM, STRING, IDENT];
SIMPTYBEGSYM:=[STRING, LPAREN, PLUS, MINUS, IDENT, INTNUM, REALNUM];
SELECTSYM:=[POINTER, PERIOD, LBRACKET];
TYPEBEGSYM:=[PLUS, MINUS, INTNUM, REALNUM, STRING, IDENT, LPAREN,
POINTER,
PACKEDSYM, ARRAYSYM, RECORDSYM, SETSYM, FILESYM];
TYPDECL:=[RECORDSYM, ARRAYSYM, SETSYM];

```

```

GETSYM;
PROGRAMHEAD([SEMICOLON]+DECLBEGSYM+STATBEGSYM);

```

```

TOP := 0;
ENTERSTDPROCS;
TOP := 1;
DISPLAY[ TOP ] := nil;

```

```

BLOCK([PERIOD]+STATBEGSYM+DECLBEGSYM);

```

```

CHECKSYM(PERIOD, 48);

```

```

if ERRCOUNT <> 0 then

```

```

begin

```

```

WRITELN; WRITELN;

```

```

WRITELN (OUTPUT, ERRCOUNT, EMES);

```

```

WRITELN (TTY, ERRCOUNT, EMES)

```

```

end

```

```

else

```

```

begin WRITELN (OUTPUT, NOERRMESS);

```

```

WRITELN (TTY, NOERRMESS)

```

```

end;

```

```

FINDCOMPS;

```

```

WRITEPINFO

```

```

end.

```

APPENDIX B  
-----

FINAL RESULT

The Strongly Connected Components of  
the Call Graph, and the Called by  
and Static Nesting information,  
for the PASREL compiler.

# LIST OF PROCEDURES AND FUNCTIONS IN THE PROGRAM

-----  
with static nesting and calls information

## BREAK

It is the root of a strongly connected component,  
which consists of the following procedure(s) :

### BREAK

It is called by the following procedures :

MAINBODY. READFILEID ENDOFLINE

## GET

It is the root of a strongly connected component,  
which consists of the following procedure(s) :

### GET

It is called by the following procedures :

NEXTCH

## MAINBODY.

It is the root of a strongly connected component,  
which consists of the following procedure(s) :

### MAINBODY.

It calls the following procedures :

WRITE	ENDOFFLINE	BLOCK	INSYMBOL	GETNEXTLIN
BREAK	WRITELN	REWRITE	READFILEID	ENTERDEBNA
ENTERUNDEC	ENTERSTDNA	ENTERSTDY		

## NEW

It is the root of a strongly connected component,  
which consists of the following procedure(s) :

### NEW

It is called by the following procedures :

ENTERDEBNA	ENTERUNDEC	ENTERSTDNA	ENTERSTDY	BODY
CASESTATEM	GETNEWGLOB	FACTOR	GETSTRINGA	DEPCST
PROCEDUREP	PARAMETERL	VARIABLEDE	TYPEDECLAR	CONSTANTDE
LABELDECLA	TYP	FIELDLIST	RECSECTION	SIMPLETYPE
COMPTYPES	CONSTANT	INSYMBOL	ERRORWITHT	

## PACK

is not called in this program.

## PAGE

It is the root of a strongly connected component,  
which consists of the following procedure(s) :

### PAGE

It is called by the following procedures :

GETNEXTLIN

## PUT

It is the root of a strongly connected component,  
which consists of the following procedure(s) :

### PUT

It is called by the following procedures :

PUTRELCOOR

## READ

It is the root of a strongly connected component,  
which consists of the following procedure(s) :

### READ

It is called by the following procedures :

NEXTCH

## READLN

It is the root of a strongly connected component,  
which consists of the following procedure(s) :

### READLN

It is called by the following procedures :

READFILEID ENDOFFLINE GETNEXTLIN

## RESET

It is the root of a strongly connected component,  
which consists of the following procedure(s) :

### RESET

It is called by the following procedures :

READFILEID

```

REWRITE
It is the root of a strongly connected component,
which consists of the following procedure(s) :
REWRITE
It is called by the following procedures :
MAINBODY.

UNPACK
It is the root of a strongly connected component,
which consists of the following procedure(s) :
UNPACK
It is called by the following procedures :
MCCODE

WRITELN
It is the root of a strongly connected component,
which consists of the following procedure(s) :
WRITELN
It is called by the following procedures :
MAINBODY. READFILEID WRITEMC WRITEHEADE WRITEFIRST
WRITEIDENT NEUZEILE INSYMBOL ENDOFLINE GETNEXTLIN
WRITERUFFE

WRITE
It is the root of a strongly connected component,
which consists of the following procedure(s) :
WRITE
It is called by the following procedures :
MAINBODY. READFILEID MCCODE WRITEHEADE WRITEFIRST
WRITEIDENT WRITEWORD SHOWRELOC NEUZEILE ENDOFLINE
GETNEXTLIN

Following procedures are declared in MAINBODY.
-----

BLOCK
It is the root of a strongly connected component,
which consists of the following procedure(s) :
BLOCK PROCEDURED
It calls the following procedures :
SKIPIFERR ERRORWITHT PROCEDURED BODY ERRANDSKIP
ERROR VARIABLEDE TYPEDECLAR CONSTANTDE LABELDECLA
INSYMBOL
It is called by the following procedures :
MAINBODY. PROCEDURED

ENDOFLINE
It is the root of a strongly connected component,
which consists of the following procedure(s) :
ENDOFLINE
It calls the following procedures :
GETNEXTLIN READLN BREAK WRITE WRITELN

It is called by the following procedures :
MAINBODY. INSYMBOL OPTIONS

ENTERDERNA
It is the root of a strongly connected component,
which consists of the following procedure(s) :
ENTERDERNA
It calls the following procedures :
ENTERID NEW
It is called by the following procedures :
MAINBODY.

ENTERID
It is the root of a strongly connected component,
which consists of the following procedure(s) :
ENTERID
It calls the following procedures :
ERROR
It is called by the following procedures :
ENTERDERNA ENTERSTDNA PROCEDURED PARAMETERL
TYPEDECLAR CONSTANTDE FIELDLIST SIMPLTYPE

```



```

020 ENTERSTONA
030 It is the root of a strongly connected component,
040 which consists of the following procedure(s) :
050 ENTERSTONA
060 It calls the following procedures :
070 ENTERID NEW
080 It is called by the following procedures :
090 MAINBODY.
100
110 ENTERSTDY
120 It is the root of a strongly connected component,
130 which consists of the following procedure(s) :
140 ENTERSTDY
150 It calls the following procedures :
160 NEW
170 It is called by the following procedures :
180 MAINBODY.
190
200 ENTERUNDEC
210 It is the root of a strongly connected component,
220 which consists of the following procedure(s) :
230 ENTERUNDEC
240 It calls the following procedures :
250 NEW
260 It is called by the following procedures :
270 MAINBODY.
280
290 ERRORWITHT
300 It is the root of a strongly connected component,
310 which consists of the following procedure(s) :
320 ERRORWITHT
330 It calls the following procedures :
340 NEW ERROR
350 It is called by the following procedures :
360 BLOCK STOREWORD MACRO FULLWORD VARIABLEDE
370 TYPEDECLAR
380
390 ERROR
400 It is the root of a strongly connected component,
410 which consists of the following procedure(s) :
420 ERROR
430 It is called by the following procedures :
440 BLOCK BODY STATEMENT WITHSTATE LOOPSTATEM
450 FORSTATEM WHILESTATE REPEATSTAT CASESTATEM IFSTATEMEN
460 COMPOUNDST GOTOSTATEM ASSIGNMENT STOREGLOBA EXPRESSION
470 SIMPLEXPR TERM FACTOR CALL CALLNONSTA
480 PROTECTION EOFEOLN PREDSUCC CHR ORD
490 ODD TRUNC SQR ABS GETLINENR
500 RELEASE MARK NEW UNPACK PACK
510 WRITEWRITE READREADLN GETSTRINGA VARIABLE GETFILENAM
520 SELECTOR LOADADDRESS MACRO INCREMENTR PROCEDURED
530 PARAMETERL VARIABLEDE TYPEDECLAR CONSTANTDE LABELDECLA
540 TYP FIELDLIST SIMPLETYPE CONSTANT SKIPIFERR
550 SEARCHID ENTERID INSYNBOL OPTIONS ERRORWITHT
560
570
580
590
600 GETBOUNDS
610 It is the root of a strongly connected component,
620 which consists of the following procedure(s) :
630 GETBOUNDS
640 It is called by the following procedures :
650 ASSIGNMENT NEW UNPACK PACK WRITEWRITE
660 SELECTOR TYP COMPTYPES
670
680 GETNEXTLIN
690 It is the root of a strongly connected component,
700 which consists of the following procedure(s) :
710 GETNEXTLIN
720 It calls the following procedures :
730 WRITE READLN WRITELN PAGE NEWPAGER
740
750 It is called by the following procedures :
760 MAINBODY. ENDOFLINE

```

```

20 INSYMBOL
30 It is the root of a strongly connected component,
40 which consists of the following procedure(s) :
50 INSYMBOL
60 It calls the following procedures :
70 INSYMBOL NEW ERROR WRITELN WRITEBUFFE
80 OPTIONS NEXTCH ENDOFLINE
90 It is called by the following procedures :
100 MAINBODY BLOCK BODY STATEMENT WITHSTATEM
110 LOOPSTATEM FORSTATEM WHILESTATE REPEATSTAT CASESTATEM
120 IFSTATEM COMPOUNDST GOTOSTATEM ASSIGNMENT EXPRESSION
130 SIMPLEEXPR TERM FACTOR CALL CALLNONSTA
140 NEW UNPACK PACK WRITEWRITE READREADLN
150 GETPUTREF GETSTRINGA VARIABLE GETFILENAM SELECTOR
160 PROCEDURED PARAMETERL VARIABLEDE TYPEDECLAR CONSTANTDE
170 LABELDECLA TYP FIELDLIST SIMPLETYPE CONSTANT
180 SKIPIFERR INSYMBOL

```

```

90 NEWPAGER
100 It is the root of a strongly connected component,
110 which consists of the following procedure(s) :
120 NEWPAGER
130 It is called by the following procedures :
140 MCCODE GETNEXTLIN

```

```

60 READFILEID
70 It is the root of a strongly connected component,
80 which consists of the following procedure(s) :
90 READFILEID
100 It calls the following procedures :
110 RESET READLN BREAK WRITE WRITELN
120 OPERAND
130 It is called by the following procedures :
140 MAINBODY

```

```

70 SEARCHID
80 It is the root of a strongly connected component,
90 which consists of the following procedure(s) :
100 SEARCHID
110 It calls the following procedures :
120 ERROR
130 It is called by the following procedures :
140 STATEMENT WITHSTATEM FORSTATEM FACTOR CALLNONSTA
150 GETINTEGER VARIABLE GETFILENAM PROCEDURED PARAMETERL
160 TYP FIELDLIST SIMPLETYPE CONSTANT

```

```

80 SEARCHSECT
90 It is the root of a strongly connected component,
100 which consists of the following procedure(s) :
110 SEARCHSECT
120 It is called by the following procedures :
130 SELECTOR PROCEDURED

```

```

50 WRITEBUFFE
60 It is the root of a strongly connected component,
70 which consists of the following procedure(s) :
80 WRITEBUFFE
90 It calls the following procedures :
100 WRITELN
110 It is called by the following procedures :
120 MCCODE MCGLOBALS INSYMBOL

```

```

50 Following procedures are declared in READFILEID
60 -----

```

```

80 OPERAND
90 It is the root of a strongly connected component,
100 which consists of the following procedure(s) :
110 OPERAND
120 It calls the following procedures :
130 SETSTATUS READCHAR READOCTAL NEXTCH
140 It is called by the following procedures :
150 READFILEID

```

Following procedures are declared in OPERAND

NEXTCH

It is the root of a strongly connected component,  
which consists of the following procedure(s) :

NEXTCH

It calls the following procedures :

READ

It is called by the following procedures :

OPERAND

READCHAR

It is the root of a strongly connected component,  
which consists of the following procedure(s) :

READCHAR

It is called by the following procedures :

OPERAND

READOCTAL

It is the root of a strongly connected component,  
which consists of the following procedure(s) :

READOCTAL

It is called by the following procedures :

OPERAND

SETSTATUS

It is the root of a strongly connected component,  
which consists of the following procedure(s) :

SETSTATUS

It is called by the following procedures :

OPERAND

Following procedures are declared in INSYMBOL

NEXTCH

It is the root of a strongly connected component,  
which consists of the following procedure(s) :

NEXTCH

It calls the following procedures :

GET

It is called by the following procedures :

INSYMBOL      OPTIONS

OPTIONS

It is the root of a strongly connected component,  
which consists of the following procedure(s) :

OPTIONS

It calls the following procedures :

ENDOFFLINE      ERROR      NEXTCH

It is called by the following procedures :

INSYMBOL

Following procedures are declared in BLOCK

BODY

It is the root of a strongly connected component,  
which consists of the following procedure(s) :

BODY

It calls the following procedures :

INSERTADDR      LEAVEBODY      NEW      ENTERBODY      ERROR

INSYMBOL      STATEMENT      WRITEMC

It is called by the following procedures :

BLOCK

```

20 COMPTYPES
30 It is the root of a strongly connected component,
40 which consists of the following procedure(s) :
50 COMPTYPES
60 It calls the following procedures :
70 GETROUNDOS NEW COMPTYPES
80 It is called by the following procedures :
90 FORSTATEME CASESTATEM ASSIGNMENT EXPRESSION SIMPLEEXPR
100 TERM FACTOR CALLNONSTA GETLINENR MARK
110 NEW UNPACK PACK WRITEWRITE READREADLN
120 GETPUTRESE GETSTRINGA GETFILENAM SELECTOR MCFILEBLOC
130 FIELDLIST STRING COMPTYPES
140
150 CONSTANT
160 It is the root of a strongly connected component,
170 which consists of the following procedure(s) :
180 CONSTANT
190 It calls the following procedures :
200 IFERRSKIP ERRANDSKIP ERROR SEARCHID INSYMBOL
210 NEW SKIPIFERR
220 It is called by the following procedures :
230 CASESTATEM FACTOR NEW CONSTANTDE FIELDLIST
240 SIMPLETYPE
250
260 CONSTANTDE
270 It is the root of a strongly connected component,
280 which consists of the following procedure(s) :
290 CONSTANTDE
300 It calls the following procedures :
310 IFERRSKIP ENTERID CONSTANT ERROR INSYMBOL
320 NEW ERRANDSKIP SKIPIFERR
330 It is called by the following procedures :
340 BLOCK
350
360 ERRANDSKIP
370 It is the root of a strongly connected component,
380 which consists of the following procedure(s) :
390 ERRANDSKIP
400 It calls the following procedures :
410 SKIPIFERR
420 It is called by the following procedures :
430 BLOCK STATEMENT LOOPSTATEM FORSTATEME FACTOR
440 CALLNONSTA GETPUTRESE PROCEDURED PARAMETERL VARIABLEDE
450 TYPEDECLAR CONSTANTDE FIELDLIST CONSTANT
460
470 IFERRSKIP
480 It is the root of a strongly connected component,
490 which consists of the following procedure(s) :
500 IFERRSKIP
510 It calls the following procedures :
520 SKIPIFERR
530 It is called by the following procedures :
540 FACTOR CALLNONSTA SELECTOR PROCEDURED PARAMETERL
550 VARIABLEDE TYPEDECLAR CONSTANTDE LABELDECLA TYP
560 FIELDLIST SIMPLETYPE CONSTANT
570
580 LABELDECLA
590 It is the root of a strongly connected component,
600 which consists of the following procedure(s) :
610 LABELDECLA
620 It calls the following procedures :
630 IFERRSKIP NEW ERROR INSYMBOL
640 It is called by the following procedures :
650 BLOCK
660
670 PROCEDURED
680 It calls the following procedures :
690 SKIPIFERR BLOCK IFERRSKIP ERRANDSKIP SEARCHID
700 PARAMETERL INSYMBOL ENTERID NEW ERROR
710 SEARCHSECT
720 It is called by the following procedures :
730 BLOCK

```

SKIPIFERR  
 It is the root of a strongly connected component,  
 which consists of the following procedure(s) :  
 SKIPIFERR  
 It calls the following procedures :  
 INSYMBOL ERROR  
 It is called by the following procedures :  
 BLOCK STATEMENT PROCEDURE PARAMETERL VARIABLEDE  
 TYPEDECLAR CONSTANTDE TYP FIELDLIST SIMPLETYPE  
 CONSTANT ERRANDSKIP IFERRSKIP

STRING  
 It is the root of a strongly connected component,  
 which consists of the following procedure(s) :  
 STRING  
 It calls the following procedures :  
 COMPTYPES  
 It is called by the following procedures :  
 EXPRESSION NEW WRITEWRITE LOADADDRESS FIELDLIST  
 SIMPLETYPE

TYP  
 It is the root of a strongly connected component,  
 which consists of the following procedure(s) :  
 TYP FIELDLIST  
 It calls the following procedures :  
 IFERRSKIP FIELDLIST GETBOUNDS TYP ERROR  
 SEARCHID INSYMBOL NEW SIMPLETYPE SKIPIFERR  
 It is called by the following procedures :  
 VARIABLEDE TYPEDECLAR TYP FIELDLIST

TYPEDECLAR  
 It is the root of a strongly connected component,  
 which consists of the following procedure(s) :  
 TYPEDECLAR  
 It calls the following procedures :  
 ERRORWITHT IFERRSKIP ENTERID TYP ERROR  
 INSYMBOL NEW ERRANDSKIP SKIPIFERR  
 It is called by the following procedures :  
 BLOCK

VARIABLEDE  
 It is the root of a strongly connected component,  
 which consists of the following procedure(s) :  
 VARIABLEDE  
 It calls the following procedures :  
 ERRORWITHT IFERRSKIP TYP SKIPIFERR ERROR  
 INSYMBOL ENTERID NEW ERRANDSKIP  
 It is called by the following procedures :  
 BLOCK

Following procedures are declared in TYP  
 -----

FIELDLIST  
 It calls the following procedures :  
 IFERRSKIP FIELDLIST CONSTANT COMPTYPES STRING  
 SEARCHID ERRANDSKIP RECSECTION TYP ERROR  
 INSYMBOL ENTERID NEW SKIPIFERR  
 It is called by the following procedures :  
 TYP FIELDLIST

LOG2  
 It is the root of a strongly connected component,  
 which consists of the following procedure(s) :  
 LOG2  
 It is called by the following procedures :  
 SIMPLETYPE

```

20 FULLWORD
30 It is the root of a strongly connected component,
40 which consists of the following procedure(s) :
50 FULLWORD
60 It calls the following procedures :
70 ERRORWITH
80 It is called by the following procedures :
90 CASESTATEM ENTERBODY PUTPAGER
100
110 GETPARADDR
120 It is the root of a strongly connected component,
130 which consists of the following procedure(s) :
140 GETPARADDR
150 It calls the following procedures :
160 MACRO5 INCREMENTR FETCHBASIS
170 It is called by the following procedures :
180 WITHSTATEM SELECTOR
190
200 INCREMENTR
210 It is the root of a strongly connected component,
220 which consists of the following procedure(s) :
230 INCREMENTR
240 It calls the following procedures :
250 ERROR
260 It is called by the following procedures :
270 ASSIGNMENT EXPRESSION FACTOR TIME RUNTIME
280 NEW UNPACK PACK WRITEWRITE GETPUTRESE
290 SELECTOR LOADADDRESS LOAD MAKECODE GETPARADDR
300
310
320 INSERTADDR
330 It is the root of a strongly connected component,
340 which consists of the following procedure(s) :
350 INSERTADDR
360 It is called by the following procedures :
370 BODY STATEMENT LOOPSTATEM FORSTATEM WHILESTATE
380 REPEATSTAT CASESTATEM INSERTBOUN IFSTATEMEN GOTOSTATEM
390 LEAVERBODY DEPCST
400
410 LEAVERBODY
420 It is the root of a strongly connected component,
430 which consists of the following procedure(s) :
440 LEAVERBODY
450 It calls the following procedures :
460 INSERTADDR MACRO3R SUPPORT MACRO3 MACRO4
470 PUTLINER
480 It is called by the following procedures :
490 BODY
500
510 LOAD
520 It is the root of a strongly connected component,
530 which consists of the following procedure(s) :
540 LOAD
550 It calls the following procedures :
560 MAKECODE INCREMENTR
570 It is called by the following procedures :
580 FORSTATEM CASESTATEM ASSIGNMENT EXPRESSION SIMPLEEXPR
590 TERM FACTOR SEARCHCODE CALL CALLNONSTA
600 PROTECTION PUTBITSTO RELEASE NEW UNPACK
610 PACK WRITEWRITE GETPUTRESE SELECTOR MAKEREAL
620
630
640 LOADADDRESS
650 It is the root of a strongly connected component,
660 which consists of the following procedure(s) :
670 LOADADDRESS
680 It calls the following procedures :
690 MACRO FETCHBASIS ERROR DEPCST MACRO3
700 STRING INCREMENTR
710 It is called by the following procedures :
720 ASSIGNMENT EXPRESSION CALLNONSTA EOFEOFNL GETINTEGER
730 MARK UNPACK PACK WRITEWRITE READREADLN
740 GETPUTRESE GETSTRINGA GETFILENAM

```

```

20 MACRO
30 It is the root of a strongly connected component,
40 which consists of the following procedure(s) :
50 MACRO
60 It calls the following procedures :
70 ERROR ERRORWIIHT
80 It is called by the following procedures :
90 FORSTATEME CASESTATEM ASSIGNMENT LOADADDRESS STORE
100 MAKECODE MACRO3R MACRO4R MACRO3 MACRO4
110 MACRO5

20 MACRO3
30 It is the root of a strongly connected component,
40 which consists of the following procedure(s) :
50 MACRO3
60 It calls the following procedures :
70 MACRO
80 It is called by the following procedures :
90 WITHSTATEM FORSTATEME CASESTATEM IFSTATEMEN GOTOSTATEM
100 ASSIGNMENT EXPRESSION SIMPLEEXPR FACTOR CALLNONSTA
110 PROTECTION EOFEOLN PREDSUCC ODD TRUNC
120 SOR TIME ABS RUNTIME PUT8BITSTO
130 RELEASE NEW UNPACK PACK WRITEWRITE
140 GETPUTRESE SELECTOR SUBLOWBOUN MAKEREAL LOADADDRESS
150 MAKECODE FETCHBASIS LEAVERODY ENTERBODY PUTLINER

20 MACRO3R
30 It is the root of a strongly connected component,
40 which consists of the following procedure(s) :
50 MACRO3R
60 It calls the following procedures :
70 MACRO
80 It is called by the following procedures :
90 LOOPSTATEM FORSTATEME WHILESTATEM GOTOSTATEM EXPRESSION
100 CALLNONSTA PREDSUCC NEW UNPACK PACK
110 SELECTOR STORE MAKECODE LEAVERODY ENTERBODY
120 SUPPORT PUTLINER

20 MACRO4
30 It is the root of a strongly connected component,
40 which consists of the following procedure(s) :
50 MACRO4
60 It calls the following procedures :
70 MACRO
80 It is called by the following procedures :
90 FORSTATEME ASSIGNMENT EXPRESSION FACTOR CALLNONSTA
100 EOFEOLN GETLINENR MARK NEW UNPACK
110 PACK SELECTOR FETCHBASIS LEAVERODY ENTERBODY

20 MACRO4R
30 It is the root of a strongly connected component,
40 which consists of the following procedure(s) :
50 MACRO4R
60 It calls the following procedures :
70 MACRO
80 It is called by the following procedures :
90 EXPRESSION PUTLINER PUTPAGER

20 MACRO5
30 It is the root of a strongly connected component,
40 which consists of the following procedure(s) :
50 MACRO5
60 It calls the following procedures :
70 MACRO
80 It is called by the following procedures :
90 ASSIGNMENT STORE MAKECODE GETPARADDR

20 MAKECODE
30 It is the root of a strongly connected component,
40 which consists of the following procedure(s) :
50 MAKECODE
60 It calls the following procedures :
70 MACRO3R MACRO MACRO5 INCREMENTR FETCHBASIS
80 DEPCST MACRO3
90 It is called by the following procedures :
100 FORSTATEME ASSIGNMENT SEARCHCODE PREDSUCC LOAD

```



```

020 PUTLINER
030 It is the root of a strongly connected component,
040 which consists of the following procedure(s) :
050 PUTLINER
060 It calls the following procedures :
070 MACRO4R MACRO3 MACRO3R PUTPAGER
080 It is called by the following procedures :
090 STATEMENT LEAVEBODY
100
110 PUTPAGER
120 It is the root of a strongly connected component,
130 which consists of the following procedure(s) :
140 PUTPAGER
150 It calls the following procedures :
160 FULLWORD MACRO4R
170 It is called by the following procedures :
180 PUTLINER
190
200 STATEMENT
210 It is the root of a strongly connected component,
220 which consists of the following procedure(s) :
230 STATEMENT WITHSTATEM FORSTATEME LOOPSTATEM REPEATSTAT
240 WHILESTATE CASESTATEM IFSTATEMEN COMPOUNDST
250 It calls the following procedures :
260 SKIPIFERR WITHSTATEM FORSTATEME LOOPSTATEM REPEATSTAT
270 WHILESTATE CASESTATEM IFSTATEMEN GOTOSTATEM COMPOUNDST
280 ASSIGNMENT CALL SEARCHID ERRANDSKIP PUTLINER
290 INSYMBOL INSERTADDR ERROR
300 It is called by the following procedures :
310 BODY WITHSTATEM LOOPSTATEM FORSTATEME WHILESTATE
320 REPEATSTAT CASESTATEM IFSTATEMEN COMPOUNDST
330
340 STORE
350 It is the root of a strongly connected component,
360 which consists of the following procedure(s) :
370 STORE
380 It calls the following procedures :
390 MACRO3R MACRO MACRO5 FETCHBASIS
400 It is called by the following procedures :
410 ASSIGNMENT GETLINENR NEW
420
430 SUPPORT
440 It is the root of a strongly connected component,
450 which consists of the following procedure(s) :
460 SUPPORT
470 It calls the following procedures :
480 MACRO3R
490 It is called by the following procedures :
500 ASSIGNMENT PREDSUCC TRUNC PAGE NEW
510 UNPACK PACK WRITEWRITE BREAK READREADLN
520 GETPUTRESE SUBLOWBOUN MAKEREAL LEAVEBODY ENTERBODY
530
540
550 WRITEMC
560 It is the root of a strongly connected component,
570 which consists of the following procedure(s) :
580 WRITEMC
590 It calls the following procedures :
600 WRITELN MCLIBRARY MCVARIOUS MCSYMBOLS MCCODE
610 MCGLOBALS MCFILEBLOC
620 It is called by the following procedures :
630 BODY
640
650 Following procedures are declared in WRITEMC.
660 -----
670
680 MCCODE
690 It is the root of a strongly connected component,
700 which consists of the following procedure(s) :
710 MCCODE
720 It calls the following procedures :
730 WRITEPAIR WRITERECOR NEWPAGER COPYCTP UNPACK
740 SHOWRELOCA WRITEWORD WRITE NEUEZEILE WRITEBLOCK
750 WRITEFIRST WRITEBUFFE
760 It is called by the following procedures :
770 WRITEMC

```



```

020 MCFILERLOC
030 It is the root of a strongly connected component,
040 which consists of the following procedure(s) :
050 MCFILERLOC
060 It calls the following procedures :
070 COMPTYPES WRITEWORD WRITEBLOCK WRITEFIRST
080 It is called by the following procedures :
090 WRITEMC
100
110 MCGLOBALS
120 It is the root of a strongly connected component,
130 which consists of the following procedure(s) :
140 MCGLOBALS
150 It calls the following procedures :
160 WRITEWORD WRITEBLOCK WRITEFIRST WRITEBUFFER
170 It is called by the following procedures :
180 WRITEMC
190
200 MCLIBRARY
210 It is the root of a strongly connected component,
220 which consists of the following procedure(s) :
230 MCLIBRARY
240 It calls the following procedures :
250 WRITEPAIR WRITEIDENT WRITEBLOCK WRITEHEADE
260 It is called by the following procedures :
270 WRITEMC
280
290 MCSYMBOLS
300 It is the root of a strongly connected component,
310 which consists of the following procedure(s) :
320 MCSYMBOLS
330 It calls the following procedures :
340 WRITEPAIR WRITEIDENT WRITEBLOCK WRITEHEADE
350 It is called by the following procedures :
360 WRITEMC
370
380 MCVARIOUS
390 It is the root of a strongly connected component,
400 which consists of the following procedure(s) :
410 MCVARIOUS
420 It calls the following procedures :
430 WRITEIDENT PUTRELCODE WRITEPAIR WRITEBLOCK WRITEHEADE
440
450 It is called by the following procedures :
460 WRITEMC
470
480 NEUEZEILE
490 It is the root of a strongly connected component,
500 which consists of the following procedure(s) :
510 NEUEZEILE
520 It calls the following procedures :
530 WRITE WRITELN
540 It is called by the following procedures :
550 MCCODE WRITEWORD
560
570 PUTRELCODE
580 It is the root of a strongly connected component,
590 which consists of the following procedure(s) :
600 PUTRELCODE
610 It calls the following procedures :
620 PUT
630 It is called by the following procedures :
640 MCVARIOUS WRITEWORD WRITEBLOCK
650
660 RADIX50
670 It is the root of a strongly connected component,
680 which consists of the following procedure(s) :
690 RADIX50
700 It is called by the following procedures :
710 WRITEIDENT

```

```

020 SHOWRELOCA
030 It is the root of a strongly connected component,
040 which consists of the following procedure(s) :
050 SHOWRELOCA
060 It calls the following procedures :
070 WRITE
080 It is called by the following procedures :
090 MCCODE WRITEWORD
100
110 WRITEBLOCK
120 It is the root of a strongly connected component,
130 which consists of the following procedure(s) :
140 WRITEBLOCK
150 It calls the following procedures :
160 PUTRELCODE
170 It is called by the following procedures :
180 MCLIBRARY MCSYMBOLS MCVARIOUS MCCODE MCGLOBALS
190 MCFILEBLOC WRITEWORD
200
210 WRITEFIRST
220 It is the root of a strongly connected component,
230 which consists of the following procedure(s) :
240 WRITEFIRST
250 It calls the following procedures :
260 WRITE WRITELN
270 It is called by the following procedures :
280 MCCODE MCGLOBALS MCFILEBLOC
290
300 WRITEHEAD
310 It is the root of a strongly connected component,
320 which consists of the following procedure(s) :
330 WRITEHEAD
340 It calls the following procedures :
350 WRITE WRITELN
360 It is called by the following procedures :
370 MCLIBRARY MCSYMBOLS MCVARIOUS
380
390 WRITEIDENT
400 It is the root of a strongly connected component,
410 which consists of the following procedure(s) :
420 WRITEIDENT
430 It calls the following procedures :
440 WRITEWORD RADIX50 WRITE WRITELN
450 It is called by the following procedures :
460 MCLIBRARY MCSYMBOLS MCVARIOUS

```

020 WRITEPAIR  
030 It is the root of a strongly connected component,  
040 which consists of the following procedure(s) :  
050 WRITEPAIR  
060 It calls the following procedures :  
070 WRITEWORD  
080 It is called by the following procedures :  
090 MCLIBRARY MCSYMBOLS MCVARIOUS MCCODE  
100  
110 WRITEWORD  
120 It is the root of a strongly connected component,  
130 which consists of the following procedure(s) :  
140 WRITEWORD  
150 It calls the following procedures :  
160 SHOWRELOCA WRITE NEUEZEILE PUTRELCODE WRITEBLOCK  
170  
180 It is called by the following procedures :  
190 MCCODE WRITERECOR MCGLOBALS MCFILEBLOC WRITEIDENT  
200 WRITEPAIR  
210

220  
230 Following procedures are declared in MCCODE  
240 -----  
250

260 CONSTRECSI  
270 It is the root of a strongly connected component,  
280 which consists of the following procedure(s) :  
290 CONSTRECSI  
300 It is called by the following procedures :  
310 COPYCSP  
320

330 COPYCSP  
340 It is the root of a strongly connected component,  
350 which consists of the following procedure(s) :  
360 COPYCSP  
370 It calls the following procedures :  
380 WRITERECOR CONSTRECSI  
390 It is called by the following procedures :  
400 COPYCTP  
410

420 COPYCTP  
430 It is the root of a strongly connected component,  
440 which consists of the following procedure(s) :  
450 COPYCTP COPYSTP  
460 It calls the following procedures :  
470 COPYCSP COPYSTP COPYCTP WRITERECOR  
480 It is called by the following procedures :  
490 MCCODE COPYSTP COPYCTP  
500

510 COPYSTP  
520 It calls the following procedures :  
530 COPYSTP COPYCTP WRITERECOR  
540 It is called by the following procedures :  
550 COPYSTP COPYCTP  
560

570 WRITERECOR  
580 It is the root of a strongly connected component,  
590 which consists of the following procedure(s) :  
600 WRITERECOR  
610 It calls the following procedures :  
620 WRITEWORD  
630 It is called by the following procedures :  
640 MCCODE COPYSTP COPYCTP COPYCSP  
650  
660

670 Following procedures are declared in STATEMENT  
680 -----  
690

700 ASSIGNMENT  
710 It is the root of a strongly connected component,  
720 which consists of the following procedure(s) :  
730 ASSIGNMENT  
740 It calls the following procedures :  
750 MACRO4 MACRO3 INCREMENTR MACRO5 LOADADDRESS  
760 SUPPORT MAKECODE GETBOUNDS STORE MAKEREAL  
770 LOAD MACRO FETCHBASIS ERROR STOREGLOBA  
780 COMPTYPES EXPRESSION INSYMBOL SELECTOR  
790 It is called by the following procedures :  
800 STATEMENT

## CALL

It calls the following procedures :

CALLNONSTA	EOFEOLN	PREDSUCC	CHR	ORD
ODD	TRUNC	SQR	ABS	TIME
RUNTIME	LOAD	EXPRESSION	PROTECTION	PAGE
PUT8BITSTO	GETLINENR	RELEASE	MARK	NEW
UNPACK	PACK	WRITEWRITE	BREAK	READREADLN
GETPUTRESE	ERROR	INSYMBOL		

It is called by the following procedures :

STATEMENT FACTOR

## CASESTATEM

It calls the following procedures :

FULLWORD	INSERTADDR	INSERTBOUN	STATEMENT	NEW
COMPTYPES	CONSTANT	INSYMBOL	ERROR	MACRO
MACRO3	LOAD	EXPRESSION		

It is called by the following procedures :

STATEMENT

## COMPOUNDST

It calls the following procedures :

ERROR	INSYMBOL	STATEMENT
-------	----------	-----------

It is called by the following procedures :

STATEMENT

## EXPRESSION

It calls the following procedures :

MACRO4R	MACRO3R	CHANGEBOOL	SEARCHCODE	STRING
MAKEREAL	ERROR	MACRO4	LOAD	COMPTYPES
INSYMBOL	LOADADDRESS	MACRO3	INCREMENTR	SIMPLEEXPR

It is called by the following procedures :

LOOPSTATEM	FORSTATEME	WHILESTATE	REPEATSTAT	CASESTATEM
IFSTATEMEN	ASSIGNMENT	FACTOR	CALL	CALLNONSTA
PROTECTION	PUT8BITSTO	RELEASE	NEW	UNPACK
PACK	WRITEWRITE	GETPUTRESE	GETSTRINGA	SELECTOR

## FORSTATEME

It calls the following procedures :

INSERTADDR	MACRO3R	STATEMENT	MACRO3	MAKECODE
MACRO	FETCHBASIS	MACRO4	LOAD	EXPRESSION
ERRANDSKIP	INSYMBOL	COMPTYPES	ERROR	SEARCHID

It is called by the following procedures :

STATEMENT

## GOTOSTATEM

It is the root of a strongly connected component, which consists of the following procedure(s) :

GOTOSTATEM

It calls the following procedures :

INSERTADDR	MACRO3	MACRO3R	INSYMBOL	ERROR
------------	--------	---------	----------	-------

It is called by the following procedures :

STATEMENT

## IFSTATEMEN

It calls the following procedures :

INSERTADDR	MACRO3	STATEMENT	ERROR	INSYMBOL
------------	--------	-----------	-------	----------

EXPRESSION

It is called by the following procedures :

STATEMENT

## LOOPSTATEM

It calls the following procedures :

ERROR	INSERTADDR	MACRO3R	ERRANDSKIP	EXPRESSION
INSYMBOL	STATEMENT			

It is called by the following procedures :

STATEMENT

## MAKEREAL

It is the root of a strongly connected component, which consists of the following procedure(s) :

MAKEREAL

It calls the following procedures :

MAKEREAL	SUPPORT	MACRO3	LOAD
----------	---------	--------	------

It is called by the following procedures :

ASSIGNMENT	EXPRESSION	SIMPLEEXPR	TERM	CALLNONSTA
------------	------------	------------	------	------------

MAKEREAL

```

0020 REPEATSTAT
0030 It calls the following procedures :
0040 ERROR      INSERTADDR      EXPRESSION      INSYMBOL      STATEMENT
0050
0060 It is called by the following procedures :
0070 STATEMENT
0080
0090 SELECTOR
0100 It is the root of a strongly connected component,
0110 which consists of the following procedure(s) :
0120 SELECTOR      EXPRESSION      SIMPLEEXPR      TERM      FACTOR
0130 CALL          CALLNONSTA      PROTECTION      PAGE      GETFILENAM
0140 PUT8BITSTO    GETLINENR      VARIABLE      RELEASE      MARK
0150 NEW           UNPACK          PACK          WRITEWRITE      BREAK
0160 READREADLN    GETPUTRESE      GETSTRINGA
0170 It calls the following procedures :
0180 SEARCHSECT    MACRO3R        MACRO4        INCREMENTR      MACRO3
0190 SUBLOWBOUND    GETBOUNDS      COMPTYPES      LOAD      EXPRESSION
0200 INSYMBOL      GETPARADDR      IFERRSKIP      ERROR
0210 It is called by the following procedures :
0220 WITHSTATEM    ASSIGNMENT      FACTOR      GETINTEGER      VARIABLE
0230 GETFILENAM
0240
0250 WHILESTATE
0260 It calls the following procedures :
0270 INSERTADDR    MACRO3R        STATEMENT      ERROR      INSYMBOL
0280 EXPRESSION
0290 It is called by the following procedures :
0300 STATEMENT
0310
0320 WITHSTATEM
0330 It calls the following procedures :
0340 STATEMENT    MACRO3        FETCHBASIS      GETPARADDR      SELECTOR
0350 ERROR        INSYMBOL      SEARCHID
0360 It is called by the following procedures :
0370 STATEMENT
0380
0390
0400 Following procedures are declared in SELECTOR
0410 -----
0420
0430 SUBLOWBOUND
0440 It is the root of a strongly connected component,
0450 which consists of the following procedure(s) :
0460 SUBLOWBOUND
0470 It calls the following procedures :
0480 SUPPORT      MACRO3
0490 It is called by the following procedures :
0500 SELECTOR
0510
0520
0530 Following procedures are declared in EXPRESSION
0540 -----
0550
0560 CHANGEBOOL
0570 It is the root of a strongly connected component,
0580 which consists of the following procedure(s) :
0590 CHANGEBOOL
0600 It is called by the following procedures :
0610 EXPRESSION
0620
0630 SEARCHCODE
0640 It is the root of a strongly connected component,
0650 which consists of the following procedure(s) :
0660 SEARCHCODE
0670 It calls the following procedures :
0680 LOAD          CHANGEOPER      MAKECODE
0690 It is called by the following procedures :
0700 EXPRESSION    SIMPLEEXPR      TERM
0710
0720 SIMPLEEXPR
0730 It calls the following procedures :
0740 MAKEREAL      COMPTYPES      SEARCHCODE      ERROR      MACRO3
0750 LOAD          TERM          INSYMBOL
0760 It is called by the following procedures :
0770 EXPRESSION

```

010 Following procedures are declared in SIMPLEEXPR  
020 -----  
030  
040

050 TERM

060 It calls the following procedures :

070 ERROR MAKEREAL COMPTYPES SEARCHCODE INSYMBOL  
080 LOAD FACTOR

090 It is called by the following procedures :

100 SIMPLEEXPR  
110

120 Following procedures are declared in TERM  
130 -----  
140

150 FACTOR

160 It calls the following procedures :

170 IFERRSKIP DEPCST MACRO4 COMPTYPES INCREMENTR  
180 NEW MACRO3 FACTOR ERROR EXPRESSION  
190 CONSTANT SELECTOR LOAD CALL INSYMBOL  
200 SEARCHID ERRANDSKIP

210 It is called by the following procedures :

220 TERM FACTOR  
230  
240

250 Following procedures are declared in SEARCHCODE  
260 -----  
270

280 CHANGEDPR

290 It is the root of a strongly connected component,  
300 which consists of the following procedure(s) :

310 CHANGEDPR

320 It is called by the following procedures :

330 SEARCHCODE  
340  
350

360 Following procedures are declared in CASESTATEM  
370 -----  
380

390 INSERTROUT

400 It is the root of a strongly connected component,  
410 which consists of the following procedure(s) :

420 INSERTROUT

430 It calls the following procedures :

440 DEPCST INSERTADDR

450 It is called by the following procedures :

460 CASESTATEM  
470  
480

490 Following procedures are declared in CALL  
500 -----  
510

520 ARS

530 It is the root of a strongly connected component,  
540 which consists of the following procedure(s) :

550 ARS

560 It calls the following procedures :

570 ERROR MACRO3

580 It is called by the following procedures :

590 CALL  
600

610 BREAK

620 It calls the following procedures :

630 SUPPORT GETFILENAME

640 It is called by the following procedures :

650 CALL  
660

670 CALLNONSPA

680 It calls the following procedures :

690 MACRO3R LOADADDRESS MAKEREAL LOAD EXPRESSION  
700 IFERRSKIP COMPTYPES SEARCHID ERRANDSKIP INSYMBOL  
710 ERROR MACRO4 MACRO3

720 It is called by the following procedures :

730 CALL

020 CHR  
 030 It is the root of a strongly connected component,  
 040 which consists of the following procedure(s) :  
 050 CHR  
 060 It calls the following procedures :  
 070 ERROR  
 080 It is called by the following procedures :  
 090 CALL

100  
 110 EOFEDLN  
 120 It is the root of a strongly connected component,  
 130 which consists of the following procedure(s) :  
 140 EOFEDLN  
 150 It calls the following procedures :  
 160 MACRO3 MACRO4 LOADADDRESS ERROR  
 170 It is called by the following procedures :  
 180 CALL

190  
 200 GETFILENAME  
 210 It calls the following procedures :  
 220 LOADADDRESS SELECTOR ERROR COMPTYPES SEARCHID  
 230 INSYMBOL  
 240 It is called by the following procedures :  
 250 PAGE GETLINENR WRITEWRITE BREAK READREADLN  
 260

270  
 280 GETINTEGER  
 290 It is the root of a strongly connected component,  
 300 which consists of the following procedure(s) :  
 310 GETINTEGER  
 320 It calls the following procedures :  
 330 LOADADDRESS SELECTOR SEARCHID  
 340

350  
 360 GETLINENR  
 370 It calls the following procedures :  
 380 ERROR STORE MACRO4 COMPTYPES VARIABLE  
 390 GETFILENAME  
 400 It is called by the following procedures :  
 410 CALL

420  
 430 GETPUTRESE  
 440 It calls the following procedures :  
 450 SUPPORT COMPTYPES MACRO3 INCREMENTR LOAD  
 460 EXPRESSION INSYMBOL GETSTRINGA ERRANDSKIP LOADADDRESS  
 470 VARIABLE  
 480 It is called by the following procedures :  
 490 CALL

500  
 510 MARK  
 520 It calls the following procedures :  
 530 ERROR MACRO4 LOADADDRESS COMPTYPES VARIABLE  
 540  
 550 It is called by the following procedures :  
 560 CALL

570  
 580 NEW  
 590 It calls the following procedures :  
 600 STORE MACRO3R SUPPORT MACRO4 MACRO3  
 610 LOAD GETBOUNDS EXPRESSION COMPTYPES STRING  
 620 CONSTANT INSYMBOL ERROR VARIABLE INCREMENTR  
 630  
 640 It is called by the following procedures :  
 650 CALL

660  
 670 ODD  
 680 It is the root of a strongly connected component,  
 690 which consists of the following procedure(s) :  
 700 ODD  
 710 It calls the following procedures :  
 720 MACRO3 ERROR  
 730 It is called by the following procedures :  
 CALL

```

750 ORN
760 It is the root of a strongly connected component,
770 which consists of the following procedure(s) :
780 ORN
790 It calls the following procedures :
800 ERROR
810 It is called by the following procedures :
820 CALL
830
840 PACK
850 It calls the following procedures :
860 MACRO3R INCREMENTR MACRO4 SUPPORT LOAD
870 MACRO3 GETBOUNDS COMPTYPES EXPRESSION INSYMBOL
880 ERROR LOADADDRESS VARIABLE
890 It is called by the following procedures :
900 CALL
910
920 PAGE
930 It calls the following procedures :
940 SUPPORT GETFILENAM
950 It is called by the following procedures :
960 CALL
970
980 PREDSUCC
990 It is the root of a strongly connected component,
000 which consists of the following procedure(s) :
010 PREDSUCC
020 It calls the following procedures :
030 MAKECODE SUPPORT MACRO3 MACRO3R ERROR
040
050 It is called by the following procedures :
060 CALL
070
080 PROTECTION
090 It calls the following procedures :
100 ERROR MACRO3 LOAD EXPRESSION
110 It is called by the following procedures :
120 CALL
130
140 PUTBITSTR
150 It calls the following procedures :
160 MACRO3 LOAD EXPRESSION
170 It is called by the following procedures :
180 CALL
190
200 READREADY
210 It calls the following procedures :
220 INSYMBOL SUPPORT ERROR COMPTYPES LOADADDRESS
230 VARIABLE GETFILENAM
240 It is called by the following procedures :
250 CALL
260
270 RELEASE
280 It calls the following procedures :
290 ERROR MACRO3 LOAD EXPRESSION
300 It is called by the following procedures :
310 CALL
320
330 RUNTIME
340 It is the root of a strongly connected component,
350 which consists of the following procedure(s) :
360 RUNTIME
370 It calls the following procedures :
380 MACRO3 INCREMENTR
390 It is called by the following procedures :
400 CALL
410
420 SQR
430 It is the root of a strongly connected component,
440 which consists of the following procedure(s) :
450 SQR
460 It calls the following procedures :
470 ERROR MACRO3
480 It is called by the following procedures :
490 CALL
500
510 TIME
520 It is the root of a strongly connected component,
530 which consists of the following procedure(s) :
540 TIME
550 It calls the following procedures :
560 MACRO3 INCREMENTR
570 It is called by the following procedures :
580 CALL

```



TRUNC

It is the root of a strongly connected component,  
which consists of the following procedure(s) :

TRUNC

It calls the following procedures :

SUPPORT MACRO3 ERROR

It is called by the following procedures :

CALL

UNPACK

It calls the following procedures :

MACRO3R MACRO4 SUPPORT MACRO3 LOAD  
GETBOUNDS INCREMENTR COMPTYPES EXPRESSION INSYMBOL

ERROR LOADADDRESS VARIABLE

It is called by the following procedures :

CALL

VARIABLE

It calls the following procedures :

SELECTOR ERROR INSYMBOL SEARCHID

It is called by the following procedures :

GETLINENR MARK NEW UNPACK PACK

READREADLN GETPUTRESE

WRITEWRITE

It calls the following procedures :

SUPPORT GETBOUNDS STRING MACRO3 COMPTYPES

INCREMENTR INSYMBOL LOADADDRESS ERROR LOAD

EXPRESSON GETFILENAM

It is called by the following procedures :

CALL

Following procedures are declared in GETPUTRESE  
-----

GETSTRINGA

It calls the following procedures :

LOADADDRESS NEW ERROR COMPTYPES EXPRESSION

INSYMBOL

It is called by the following procedures :

GETPUTRESE

Following procedures are declared in ASSIGNMENT  
-----

STOREGLOB

It is the root of a strongly connected component,  
which consists of the following procedure(s) :

STOREGLOB

It calls the following procedures :

ERROR STOREWORD GETNEWGLOB

It is called by the following procedures :

ASSIGNMENT

Following procedures are declared in STOREGLOB  
-----

GETNEWGLOB

It is the root of a strongly connected component,  
which consists of the following procedure(s) :

GETNEWGLOB

It calls the following procedures :

NEW

It is called by the following procedures :

STOREGLOB

STOREWORD

It is the root of a strongly connected component,  
which consists of the following procedure(s) :

STOREWORD

It calls the following procedures :

ERRORWITHT

It is called by the following procedures :

STOREGLOB

\*\*\*\*\*

A63792

EE-1980-M-SEO-OVE